Control and selection techniques for the automated testing of reactive systems

DOI:
10.6100/IR581420

Document status and date:
Published: 01/01/2004

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Control and Selection Techniques for the Automated Testing of Reactive Systems

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Goga, Nicolae

Control and selection techniques for the automated testing of reactive systems / by Nicolae Goga. -
Eindhoven: Technische Universiteit Eindhoven, 2004.
Proefschrift. - ISBN 90-386-0892-6
NUR 992
Subject headings : automatic test methods / reactive computer systems /
statistical tests / metric spaces
CR Subject Classification (1998) : B.4.5, D.2.5

The work in this thesis has been carried out under the auspices of the research school IPA
(Institute for Programming research and Algorithmics).

IPA dissertation series 2004-9

Printed by University Press Facilities, Eindhoven
Cover design by Paul Verspaget

CONTROL AND SELECTION TECHNIQUES FOR THE AUTOMATED TESTING OF REACTIVE SYSTEMS

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR AAN DE TECHNISCHE UNIVERSITEIT EINDHOVEN, OP GEZAG VAN DE RECTOR MAGNIFICUS, PROF.DR. R.A. VAN SANTEN, VOOR EEN COMMISSIE AANGEWEZEN DOOR HET COLLEGE VOOR PROMOTIES IN HET OPENBAAR TE VERDEDIGEN OP DONDERDAG 7 OKTOBER 2004 OM 16.00 UUR

DOOR

NICOLAE GOGA

GEBOREN TE BOEKAREST, ROEMENIË

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. L.M.G. Feijs
en
prof.dr. H. Brinksma

Copromotor:
dr. S. Mauw

Acknowledgments

In 1998 I was hired as a PhD student in the Côtes-de-Resyste project, a project in the area of automatic test generation funded by the Dutch Technology Foundation STW. Having reached the end of Côtes-de-Resyste and the end of my work for writing my thesis, I would like to thank everybody who made it possible for me to write this thesis. First of all, I would like to thank my direct supervisors Loe Feijs and Sjouke Mauw. Without their help and advice I would not have gotten very far. Secondly, I would like to thank Jan Tretmans, Ed Brinksma and all the other members of the CdR team for their supervision, advice and cooperation in the time when CdR was running and the thesis was written.

I am also thankful to Judi Romijn for offering me a position as a postdoc in the project which she is leading, for accepting to work on my thesis in the time which I was supposed to work only for her project and for her continuous support offered to me. My thanks goes also to Jos Baeten whose help during the years contributed to the advance of my scientific career.

Wan Fokkink and Jens Grabowski are kindly thanked for their willingness to take part in the PhD committee.

At Technische Universiteit Eindhoven, I have been already working six years in which I enjoyed a pleasant atmosphere and nice social contacts.

My final thanks go to my parents and my wife for their continuous support, love and understanding.

STELLINGEN

for PhD thesis

Control and Selection Techniques for the Automated Testing of Reactive Systems

by

Nicolae Goga

1. Assume that there is a large number of tests executed against an implementation. In this case, the generation based on test purposes [1] compared to the UIO test generation methodology will find in most of the cases a larger number of errors.

   [1] Chapter 3, this thesis

2. The introduction of probabilities into test generation and execution [2] can increase the effectiveness of a test suite.

   [2] Chapter 6, this thesis

3. The measures for test coverage proposed in the literature are based on heuristics. The choice for a specific coverage measure relies on the preferences of an individual, group or institution for one heuristic or another.

4. If one says that "This implementation is 99% free of errors", there is no generally agreed way to measure that.

5. For black box testing, using a *Cycling* heuristic for generating tests requires making use of non-available information in an implementation, namely the state in which the implementation is at a given moment of time. The *Cycling* strategy [3] is based on assumptions related to the automaton model of the implementation (for example, the assumption that the implementation is a minimal automaton).

[3] Chapter 8, this thesis

6. It is expected that sending information by means of a number of smaller messages instead of packaging it in one big message reduces the waiting time overhead for processes that work in parallel on different machines. Experience shows that this is not true in all cases [4].

[4] N. Goga, Z.Racovita, and A. Telea. Texture mapping in a distributed environment. In E. Banissi, C. Chen, and G. Clapworthy, editors, *7th conference on Information Visualization (IV'03)*, volume 7, pages 36–41. IEEE CS Press, 2003.

7. The common belief is that international standards in the computer network area are free from errors that could be found through simulation and verification. Unfortunately this not true [5].

[5] A. J. Mooij, N. Goga, W. Wesselink, and D. Bošnački. An analysis of medical device communication standard IEEE 1073.2. In C. E. P. Salvador, editor, *Communication Systems and Networks*, pages 74–79. IASTED, ACTA Press, 2003.

8. The evolution theory does not explain 100% the origin and the diversity of life yet. Some examples: no living cell could be obtained experimentally from non-organic substances; certain intermediate fossil links are missing.

9. Choosing music is not a simple matter of taste. If one wants to be healthy, one needs to be careful concerning which music one is listening to.

10. Eating vegan increases life expectancy (average duration).

11. The current tendency is that USA will become the dominant super power of the world.

Contents

Chapter 1

Introduction

Testing is an important activity carried out in the development cycle of every product. Starting from its design till its physical realization, in every stage, tests are performed and their results are used for the further improvements of the product. Therefore, the testing activity is very important and it receives a lot of attention from industry and academia. Especially, the care with which a test is designed and performed is essential for the quality of the test results.

There are explosive growth domains which seem to have completely changed the human life in the last years. For example, the computer which is part of each reactive system is penetrating everywhere. A child who is only two or three years old is playing games on a computer or a computer network. The postal mail is replaced by electronic mail. In medicine computers help in better diagnosis and treatment of diseases. Internet is essential today for everybody. These are only some uses which reveal that computers are vital for our daily life. Other examples, taken from the area of reactive systems which are important for us today, are: consumer electronic products, such as VCRs or TVs, or bank cash machines.

We will refer to some examples which show the importance of testing for reactive systems.

- The first example is taken from the computer field. Starting in 1996, an industrial experiment was carried out by INRIA Rhône-Alpes and BULL engineers concerning the design and testing of POLIKID [GVZ00], a CC-NUMA (Cache Coherent – Non Uniform Memory Access) multiprocessor architecture developed by BULL. Due to performance reasons, the BULL engineers designed their own version of the CC-NUMA cache coherency protocol for the multiprocessor POLIKID. The experiment was about the design of the cache coherency protocol and testing it on a working prototype of a POLIKID machine. In the design phase, the formalization of the specification and the verification activities revealed 20 serious issues, among which 8 behavioural errors in the coherency protocol. In the testing phase, the use of TGV [JM99], a tool for automatic test generation, led to the discovery of 5 bugs on the working prototype of POLIKID. These bugs could not have been caught by hardware testing experts using traditional technology. The results obtained in this experiment were rewarded with a BULL Eureka Research and Development Award.

- The second example shows that the testing activity is beneficial also in the field of consumer electronics. This example reveals that errors can occur everywhere and that the testing activity can help to detect them. The EasyLink protocol was first defined in 1996 by Philips Research to facilitate communication between a TV and one or more VCRs. Recently, a testing activity was carried out inside Philips to check out the consistency of this protocol [BFHdV01]. The tests led to the discovery of an unwanted behaviour, which was caused by lack of clarity of the EasyLink protocol specification. The conclusion was that, from a protocol conformance perspective, the unwanted behaviour is an error because, as a result of it, the TV could end up in a state with a blue screen and the only way to escape from this state is to turn the TV off and on again.

- The third example shows a disastrous event which could be prevented by a more careful test activity. This example is from the aerospace domain which uses reactive systems, such as computerized positioning systems. On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after its lift-off from Kourou French Guiana. The costs of the experimental equipment was enormous. The rocket was on its first voyage, after a decade of development costing \$7 billion. Together with Ariane 5, Cluster – a \$500 million set of four scientific satellites – was also destroyed. The press was quickly involved by reporting the disaster (for example, see The New York Time Magazine of 1 December 1996). Over the following days an Enquiry Board led by Prof. Jacques-Louis Lions (Academie des Sciences, France) was set up to determine the causes of the launch failure. Quoting from the report, the causes of the explosion were:

  > The failure of Ariane 501 was caused by the complete loss of guidance and altitude information 37 seconds after start of the main engine ignition sequence (30 seconds after lift-off). This loss of information was due to specification and design errors in the software of the inertial reference system.
  >
  > The internal SRI software exception was caused during execution of a data conversion floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than could be represented by a 16-bit signed integer.

  In the above quote, SRI stands for Système de Référence Inertielle or Inertial Reference System. The designers of the SRI of Ariane 5 were re-using code from the SRI of Ariane 4. When re-using it, the designers did not consider adequately the change of some flight parameters which had other ranges for Ariane 5 than for Ariane 4. This led to the error of the conversion which started a chain of events which eventually caused the explosion. The report concluded also that a more carefully carried out test activity could have discovered this error.

Each of these examples shows the importance of testing from different perspectives. The first one reveals that even if a careful design activity is done using formal methods, testing can lead to the discovery of bugs. This example concerns a multiprocessor architecture. The second one, taken from the field of consumer electronics, shows that the testing activity is beneficial also in this domain. The third one describes a disastrous event from the field of the aerospace domain which could be averted if testing was done more carefully. Other similar examples can easily be found, for example see http://www.bugnet.com/, but we will stop here.

Before going deeper in testing, we should mention that there are complementary approaches to the problem of software correctness. Preventing errors is better, in terms of costs associated, than introducing them first, trying to find and repair them later. For this reason, methodologies have been developed which aim at preventing errors, such as correctness proofs ([Zwi88]). But even when such methods are employed, the errors can still occur (see also the example regarding the CC-NUMA cache coherency protocol). Therefore, these methods are useful, but the industry still considers testing indispensable. The conclusion is that each of these approaches has his own importance and that testing is a worthwhile activity.

In Section 1.1 we will enter in more details regarding the testing of reactive systems. Section 1.2 presents a number of terms and developments in this area. Section 1.3 identifies issues which are subject of research and improvements in this domain. Section 1.4 describes the structure of the thesis.

1.1 Conformance testing of reactive systems

When we discussed the importance of computers, we mentioned the term *reactive systems*. What is a reactive system? The common understanding [Hee98] is that a reactive system is a system which exchanges information with its environment. It receives stimuli from the environment, which are commonly denoted as inputs, and it sends responses to stimuli, which are referred to as outputs. The system is expected to react in a timely manner. Many systems can be viewed as reactive systems. Some examples are computers or phones in a network and hardware components which interact with each other. Important examples of reactive systems are process control systems, communication protocols and embedded-software systems. But not all information processing systems are considered to be reactive systems. One counter-example is a computation process which writes the results to a file system (a batch-oriented computation). This process for which the main task is to compute, cannot be regarded as a reactive system because there is no continuous interaction with its environment nor are there any hard time constraints.

In the examples which we gave regarding the testing of reactive systems we showed that their testing is important for the industrial community. But this testing is also important for academia because there are problems in testing which are not well understood and which are researched by academia. The joint effort of these communities built up more standards on this domain. In this chapter we will refer to two standards which deal with testing and which come from the telecom area: OSI IS-9646 [ISO92] and ITU-T Z.500 Recommendation ‘Framework: Formal Methods in Conformance Testing’ [IT97]. These two standards will be discussed in more detail in Section 1.2. Another indication that the topic is of interest is the list of industrial and prototype tools which are developed for testing reactive systems. We will mention only some names which are: Autolink [SKGH97], Phact [FMMvW98], TGV [JM99] and TorX [BFdV$^+$99]. Moreover these tools are using specification languages that are especially designed for the formalization of the behaviours of such systems and which are themselves subject of standardization. Important specification languages are: SDL [CCI92], Estelle [ISO89a], LOTOS [ISO89b], MSC [IT93] and TTCN [ISO92]. And not at the last place is the ongoing research done in this area. A large number of articles and PhD theses from this domain can easily be found. Some examples are the research presented in the PhD theses [Vra98] and [Kwa97].

The specification languages already mentioned give us the opportunity to concentrate a little bit more on some elements which play a role in testing. First, we have a specification in a specification language which describes the allowed behaviours of the system to be tested. Second, the implementation denotes the material realization of the system. Now, the objective of testing is to find out if the implementation conforms to the abstract description, the specification. This approach to the testing of a reactive system is called conformance testing, which means that an implementation should conform to its specification. We will say more about conformance testing in the next section.

Now let us introduce another aspect of reactive system testing. The common practice today is automation. This practice is justified by the fact that a manual process is time consuming and expensive. In particular this applies for testing. Automatic testing means that the tests are derived from the specification by a test generation algorithm and executed against the implementation. In the area of automatic testing, the project Côte-de-Resyste (COnformance TEsting of REactive SYSTEms, CdR by short), funded by the Dutch Technology Foundation STW, was born as a collaboration between two universities, the University of Twente and the Eindhoven University of Technology in The Netherlands, and two industrial companies, Philips and KPN. Later KPN left the CdR project and another industrial company, Lucent, joined the project. CdR, under which auspices this thesis was written, aims at developing methods, techniques and tools for testing reactive software systems. The CdR project is only one example which shows that automatic testing is of interest today.

Before ending this section we want to introduce another issue which is subject of ongoing research and which takes a central position in this thesis. Once one has a specification and starts to automatically derive tests, the set of tests which can be derived is usually large, even infinite. Therefore, the selection of an appropriate set of tests with a large capability to detect errors is of great importance. Some examples of research in this area are presented in [Groz] and [Vuong].

1.2 Conformance testing: terms and developments

In this section we describe some key elements presented in two standards OSI IS-9646 and ITU-T Z.500 Recommendation 'Framework: Formal Methods in Conformance Testing'. This will give the reader the opportunity to get a better understanding of the conformance testing of reactive systems and of the topics addressed in this thesis. The standard OSI IS-9646 'OSI Conformance Testing Methodology and Framework' [ISO92] defines a methodology and framework for protocol conformance testing. Originally it was made for OSI protocols. Later it was used also for other kinds of protocols (for example, the CC-NUMA cache coherency protocol). The following parts of this standard are of interest for us:

1. General Concepts;

2. Abstract Test Suite Specification;

3. Tree and Tabular Combined Notation (TTCN);

We will concentrate on part 1) and 2) which discuss general concepts of protocol conformance testing. We will briefly present part 3) and we will acknowledge the recent developments of TTCN.

The introduction of OSI IS-9646 part 1) states that conformance testing can not guarantee conformance to a given specification. Why does it say this? Suppose no errors were found during testing. Then the strongest statement we can make is precisely that: 'No errors were found', but this does not necessarily mean that the implementation does not contain any errors. The main reason is that conformance testing can never be comprehensive enough to guarantee error-free behaviour because the number of tests required will be too large. However, this should not lead one to believe that conformance testing is not a worthwhile activity. Errors can be revealed through testing and then repaired. The conformance testing does not guarantee that no errors exist in the implementation, but it gives confidence that an implementation has the required behaviours described by its specification.

One interesting observation which can be made concerns the relation between conformance testing and *interoperability testing*. OSI IS-9646 chooses to test in isolation an entity which participates in a protocol exchange. It does not require to test a whole assembly of entities which are engaged in a communication. Testing more entities which communicate with each other is called interoperability testing. Performing an interoperability test will check the conformance of the whole assembly viewed as a black-box against a service specification. The reason for the choice made in OSI IS-9646 is that it is cheaper to test each entity in isolation than to test all the possible combinations for the whole network of entities. For example, let us consider the situation of a network of 2 similar entities and

5 vendors which can provide entities. The entities are similar in the sense that they are supposed to implement a common specification. Testing each entity in isolation will cost 5 units (a unit can be an average set of tests) because there are 5 different vendors, each of them being able to provide entities. Now let us consider the situation of an interoperability test performed for this network of entities. For reducing the number of tests, it can be assumed that when connecting two entities provided by the same vendor, the network as a whole works correctly and there is no need for testing. This assumption usually does not hold if the similar entities are provided by two different vendors. Therefore an interoperability test is needed for checking the conformance of the whole assembly in this case. For a network of 2 entities and 5 possible vendors for them, the costs for the interoperability testing is 10 units, which come from the general formula of $\frac{1}{2} \times n \times (n-1)$ for a network of 2 entities and n vendor companies. The cost of the interoperability testing is twice higher than the one of testing an entity in isolation. For such reasons, the standard chooses to test an entity in isolation. In theory, if each entity conforms to its specification, a network of entities should function correctly. There are subtle reasons why this is not always the case in practice. One reason is the fact that, as OSI IS-9646 states, the conformance can not be guaranteed in all cases. Another reason is that there are many incompatible options which might be implemented by the entities. Therefore, the interoperability testing is still useful to be performed (we will come back to this discussion).

OSI IS-9646 part 1) introduces the concepts of the testing process and its constituent parts. The process is outlined in Figure 1.1. Presenting this process gives the reader the possibility to get acquainted with elements which play roles in testing activities. Moreover this testing process is often referred to as being one of the starting points for the test generation tools with which we are working in this thesis. Therefore we considered it worthwhile to outline it. We will start to introduce its component elements from the figure step by step.

The specification of the protocol is given in natural language by means of so called *conformance requirements*. The conformance requirements of the protocol are classified in two categories: 1) static conformance requirements and 2) dynamic conformance requirements.

The *static conformance requirements* are 'limitations on the combination of implemented capabilities which are permitted' for conformance. An example of static conformance requirements are clauses 14.3 and 14.4 of IS-8073, the OSI Transport Protocol.

14.3 If the system implements Class 3 or 4, it shall also implement Class 2.
14.4 If the system implements Class 1, it shall also implement Class 0.

The *dynamic conformance requirements* are the actual protocol requirements, that is, the requirements that concerns input/output behaviours of the protocol entity itself.

Based on the static and the dynamic conformance requirements a checklist with the capabilities of the system to be tested is produced. This checklist is called *Protocol Implementation Conformance Statement Proforma* (PICS Proforma) and it is signed and delivered with every implementation of a standard protocol. The PIXIT (*Protocol Implementation eXtra Information for Testing*) contains system-specific information outside of the scope of the base standard. An example of such information might be the particular system's addresses to be used during testing or the precise value of a parameter within a permitted range of values. Unlike the PICS Proforma which is part of the base standard, the PIXIT is supplied by the tester on the basis of details provided by the implementer (the tester makes a PIXIT Proforma; the implementer willing to have his IUT tested completes it; then it is called PIXIT).

The first part of the testing process, *Static Conformance Review*, involves the examination of the PICS. This is a process of comparing what the implementer says has been implemented against what the base standard says should have been implemented in the particular *Implementation Under Test*

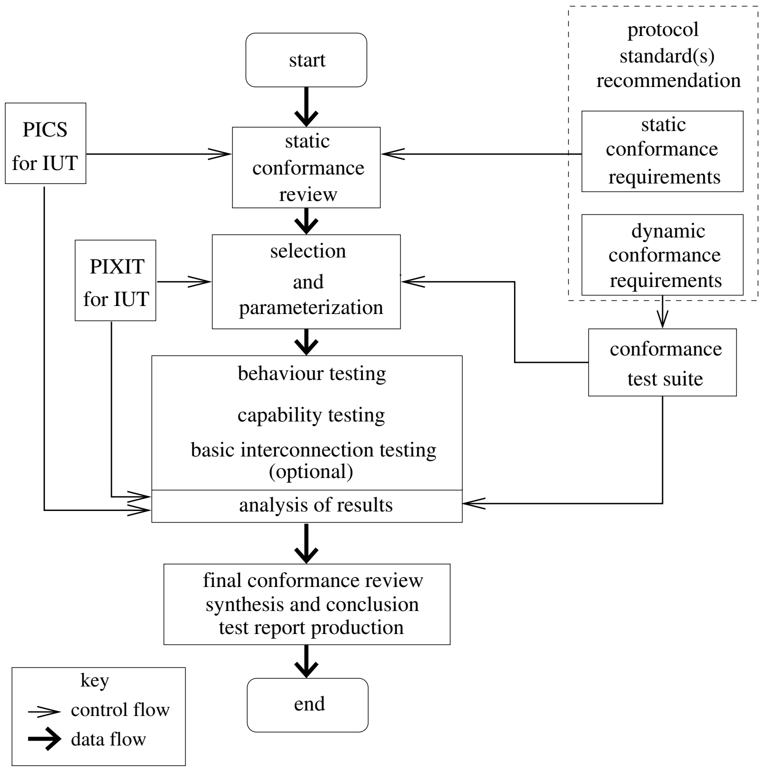

start
PICS for IUT
static conformance review
protocol standard(s) recommendation
static conformance requirements
dynamic conformance requirements
PIXIT for IUT
selection and parameterization
conformance test suite
behaviour testing
capability testing
basic interconnection testing (optional)
analysis of results
final conformance review synthesis and conclusion test report production
end
key
control flow
data flow

Figure 1.1: Testing process overview.

(IUT). The next step, *Selection and Parameterization*, is to establish the set of tests that match the characteristics of the IUT. These tests are drawn from the *Conformance Test Suite*, which contains theoretically a complete set of tests covering every aspect of the base standard. The assumption that the Conformance Test Suite is complete is a pragmatic point of view taken by OSI IS-9646, because, as we already said, this is not usually possible in practice (commonly, the completeness implies a large, even infinite number of tests). Based on the set of selected tests, three sorts of dynamic tests can be performed:

1. Behaviour Tests;
2. Capability Tests;
3. Basic Interconnection Tests.

Behaviour Tests are the dynamic tests performed. *Capability Tests* are concerned with specific test statements made in the PICS about specific parameters. If, for example, the PICS indicates that in a transport protocol sizes of 256 and 512 octets are supported, then the tests would be appropriately parameterized to check this claim. *Basic Interconnection Tests* are optional. They are a subset of the complete set of behaviour tests contained in the Conformance Test Suite. The motivation for these tests is that rudimentary testing up front might indicate whether or not it is worth embarking on the full test campaign.

The *Analysis of Results* means the analysis of the tests results. A *test suite* for a given protocol contains a number of different discrete tests. For each test there will be a result, or *outcome*. For every outcome, that is, for every individual test performed, a test verdict will be assigned, either **pass**, **fail** or **inconclusive**. A **pass** verdict means that the observed test outcome provides evidence of conformance to the conformance requirement under consideration. A **fail** verdict means that the observed test outcome demonstrates nonconformance with respect to the conformance requirement under consideration. An **inconclusive** verdict means that there is insufficient evidence for either **pass** or **fail** verdict to be assigned, i.e. the test objective (purpose – see below the description of test purposes in TTCN) is not fulfilled but no conformance requirement is invalidated by the test outcome.

The final stage of the process, *Final Conformance Review*, is the release of two test reports:

1. System Conformance Test Report (SCTR);
2. Protocol Conformance Test Report (PCTR).

In general, one or more protocols may be tested within a given system. Thus there is a PCTR for each protocol and a SCTR for the overall system. The SCTR contains a summary of the test results and the PCTR contains a detailed list of all the tests executed and the results (verdicts) for each of them.

These are the ingredients of the testing process as it is presented in OSI IS-9646. Many of these ingredients play a role in the research described in this thesis. We will give some examples. The first example is the Selection and Parameterization phase which exists in the experiments presented in this thesis, but in a more advanced form. The Parameterization usually appears in the same form as described by OSI IS-9646 in which the tester chooses by himself specific values for parameters and variables. But the Selection is updated from a manual form to an automatic form through the use of a test generation algorithm. Another example is the use of the test verdicts, **pass**, **fail** and **inconclusive**, throughout the whole content of the thesis.

It is interesting also to consider the manner in which testing can be performed, considering the structure of a layered and distributed system. OSI IS-9646 presents four *Abstract Test Methods*:

1. Local Test Method;
2. Distributed Test Method;
3. Coordinated Test Method;
4. Remote Test Method.

For giving an idea what these Abstract Test Methods look like we will present the *Remote Test Method*. The abstract model for the Remote Test Method is shown in Figure 1.2.

The Remote Test Method makes no assumptions about the internal design of the *System Under Test* (SUT) or the IUT within. The tester is distributed in two parts: *Upper Tester* (UT) and *Lower Tester* (LT). It is necessary to coordinate the actions of the Upper and the Lower Tester. This is necessary to ensure that events are synchronized properly so that before and after effects, at the upper and lower boundaries can be analyzed correctly. This coordination is performed by the *Test Coordination Process* (TCP). In the figure we see that there is a notional Upper Tester, as well as a notional Test Coordination Process. The notional UT represents the fact that there may be some methods of driving the IUT. If there is, however, the means are not standardized and are implementation-dependent. Furthermore, the extent of control that can be exerted over the SUT/IUT is extremely limited. In the figure we can also see that service primitives can be sent and received via the layer service provider at designated *Points of Control and Observation* (PCOs). The tester and the IUT can also exchange

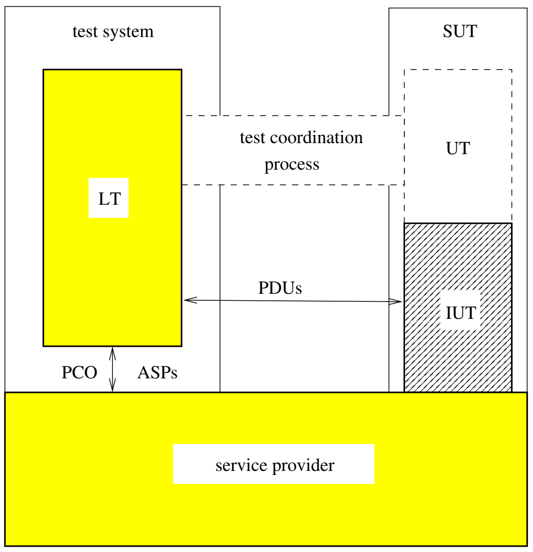


Figure 1.2: Remote Test Method.

Protocol Data Units (PDUs). To distinguish this architecture from what one might meet in practice, this architecture is called *abstract* and the service primitives are denoted also as *Abstract Service Primitives* (ASPs).

The other three methods are built in a similar style as the Remote Test Method, using the same building elements which were used for the Remote Test Method (PCO, LT, UT, etc.). These methods are designed to accommodate the variation in the control and observation that can be achieved over the SUT, but in practice variants of these methods are used. In this thesis, one variant of these methods is used in the benchmarking experiment with the Conference Protocol case study.

The testing activity is carried out by executing tests. A *Test Suite* is built up in a nested hierarchical fashion (see Figure 1.3) from a number of named test units. The smallest indivisible unit of a Test Suite is the *Test Event*. The next incremental unit, up from the Test Event, is the *Test Step*. A Test Step is comprised of a number of Test Events. A *Test Case* is the main fundamental building block of the Test Suite. A Test Case is the unit that performs a particular test, corresponding to a particular feature of the protocol under test. A *Test Purpose* is a textual description of the objective of a particular test. The verdict obtained by executing the Test Case can be **pass**, **fail** or **inconclusive**. A *Test Group* is a grouping of Test Cases. The Test Suite is the highest level. As Figure 1.3 shows, it can range from many Test Groups of many Test Cases containing many Test Steps and Test Events, to a very simple case of a single Test Case with a single Test Event.

The third part of OSI IS-9646 defines the test case description language *Tree and Tabular Combined Notation* (TTCN), a language suitable for describing Test Suites. In a few words, as indicated by the name Tree and Tabular Combined Notation, a TTCN test suite is a collection of different tables. The information of the tables is organised into tree structures which have hierarchies similar to the nested hierarchical structure from Figure 1.3. All objects of a test suite, e.g. PDU, ASPs, test cases or test steps, are specified in tables. In this thesis we are using the OSI IS-9646 version of TTCN together with the nested hierarchical structure of the test suite in the experiment with Autolink on the Conference Protocol case study. Also, we are using in our testing theory tree structures for test cases (trees represented in a labelled transition systems formalism [Arn93]).

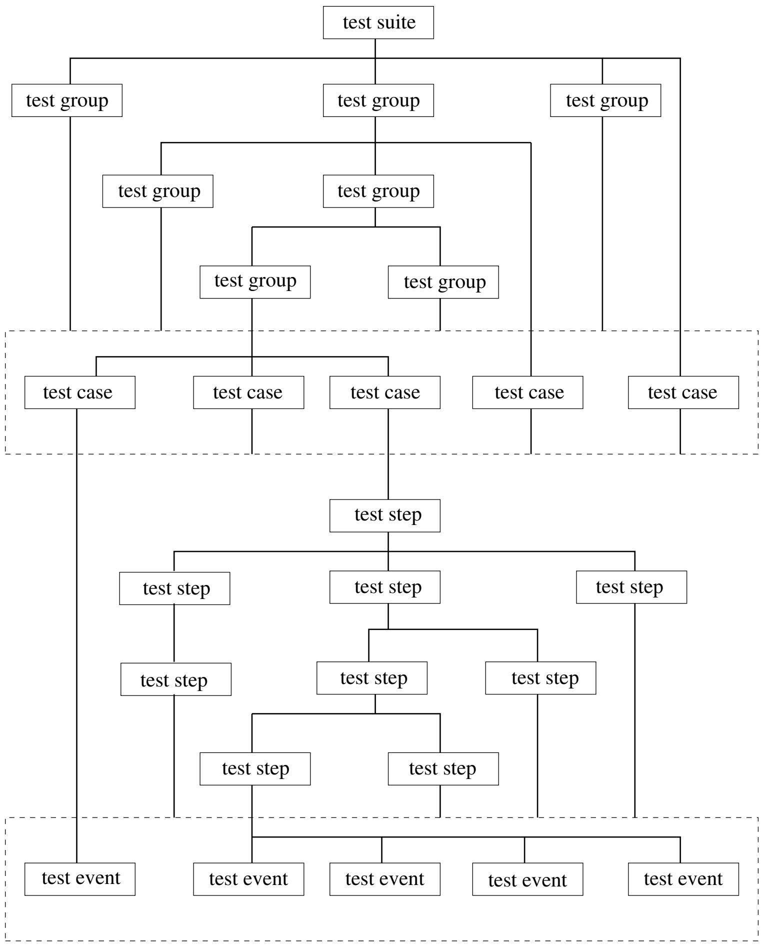


Figure 1.3: Test Suite structure.

Recently a new version of TTCN is in the process of being defined [BRS01]. Although we are not using it in this thesis, we found it useful to enlarge the horizon of this thesis by presenting recent developments of TTCN. This will give the reader the possibility to have a quick view to other research lines in the domain of the conformance testing of reactive systems. We will try to outline key elements of the new TTCN. TTCN is now an abbreviation for *Testing and Test Control Notation*, an abbreviation which reflects the characteristics of the new version. The new TTCN is in a programming-like style with flexible data support and several presentation formats. This style replaces the tabular style. One of the improvements is the module concept. Modules are the building blocks of all the test suites written in the new version of TTCN. In a module, basic programming statements may be used to select and control the execution of the test suite, a thing which was not possible in the old version of TTCN. This improvement shows that the main topics of this thesis, respectively selection and control of testing, were important issues also for the new version of TTCN. The new TTCN offers programming constructs for selection and control while we work towards automation of selection and control. The programming style allows also the introduction of new concepts such as external functions or data. One type of presentation is the graphical format which is an extended MSC. We can remark that the work done by the Formal Methods TU/e group in establishing MSC as a standardized language is showing again its usefulness, in this context of a new version of TTCN.

Another development line from OSI IS-9646 was the ITU-T Z.500 Recommendation 'Framework: Formal Methods in Conformance Testing', which is a standard supplementary to OSI IS-9646. We found it interesting to extract some elements from it which add more understanding to the things presented up to this point in this chapter.

While in OSI IS-9646 the specification was supposed to be described in natural language, this new standard makes a step ahead in formalization by assuming that the specification prescribes the behaviours of a system using a formal description technique (such as SDL or LOTOS). The set of specifications is denoted by *SPECS*. About the implementation the reasoning is as follows [IT97]:

> A specification is a formal object while an implementation is a physical one. In order to formalize the concept of conformance, these different kinds of objects have to be related. Implementations can not be subject to formal reasoning as they are not formal objects. Therefore it is not possible to define directly a formal relation between implementations and specifications.
>
> In the remaining of this document, it is assumed that any implementation IUT [...] can be modeled by an element m_{IUT} in a formalism *MODS* (e.g., labeled transition systems, finite state machines). [...]. The activity of testing consists of extracting information from IUT by testing it, such that from this information the model m_{IUT} can be constructed in sufficient detail to decide conformance about it.

The conformance is defined in the context of an implementation relation $\underline{imp} \subseteq MODS \times SPECS$. An implementation IUT conforms to a specification s with respect to the relation $\underline{imp}$ if $m_{\mathrm{IUT}}\underline{imp}\ s$. ITU-T Z.500 does not require to test an entity which participates in a protocol exchange in isolation, as OSI IS-9646 requires. For this supplementary standard, an IUT can be an entity or a whole assembly of entities viewed as a black-box against a service specification. For this reason, interoperability testing is seen as a special case of conformance testing. In this thesis we adopt the same view as ITU-T Z.500 regarding interoperability testing (we see it as a special case of conformance testing).

As OSI IS-9646 already said in its introduction, when testing, one can not obtain certainty concerning conformance between implementation and specification. In the context of an implementation relation this can be expressed more precisely by defining properties assigned to a test suite. The first one defined is the *exhaustiveness*: a test suite is exhaustive if the set of all models that pass the test suite is a subset of the set of conforming models. The second one is the *soundness*: a test suite is sound if the set of conforming models is a subset of the set of models that pass the test suite. The last one is the *completeness*: a test suite is complete if it is both sound and exhaustive, that means, the set of conforming models equals the set of models that pass the implementation. It is remarked also that in general it is not possible to construct a finite exhaustive test suite (the testing activity can not guarantee conformance) although it would be nice to have a complete test suite.

The standard addresses also the topic of automatic test generation. The test generation is seen as a function which provides a test suite from a specification described using a formal description technique. The generated tests are required to be sound. Also, for reducing the size of a test suite, different test suite size reduction strategies are identified and described at a high level of abstraction. And for expressing the quality of a test suite in terms of its error-detecting capabilities, a *coverage* measure is indicated to be used. In this thesis all the elements which we presented from the ITU-T Z.500 Recommendation, e.g. *SPECS*, $\underline{imp}$, exhaustiveness and soundness, play important roles throughout the whole content.

The things described up to this point give a general idea about conformance testing. We should remark that there are also other kinds of testing. We can mention for example *performance testing* which is about measuring the quantitative aspects of an implementation or *reliability testing* which means to test whether the implementation works correctly during a long period of time. This thesis presents only research in the area of conformance testing and does not treat other kinds of testing. There are the following levels of accessibility of an SUT in testing. The *White-box* testing means that the tester has access to the internal structure of the system under test. The *Grey-box* testing means that the tester has access to some part of the system under test. The *Black-box* testing means that the tester has no access to the internal structure of the system under test. It can be said that the black and white testing are the extremes and the grey-box testing is for everything in between. In this thesis, the system under test is checked in a black-box style.

1.3 Open issues in conformance testing, limitations and goals

In a previous section we mentioned that the research presented in this thesis was done within the CdR project. In the CdR project proposal (for example, see http://fmt.cs.utwente.nl/CdR/) several open issues were identified as being worthwhile to be researched by the CdR team. Consequently, this thesis will address some of these issues. Although we already presented some open problems which exist in this area and which were also topics of research for CdR, we will try to describe them again in more detail.

An open issue is the comparison between existing tools on the area of automatic test generation. There are several tools for test generation using different techniques. We mentioned already some names such as Autolink or TGV. From a user point of view, having to select one of them for use, it is interesting to know what the possibilities and limits of these tools are. Therefore comparing these tools from different perspectives, such as speed or error detection power, is very useful for the testing community. At the same time, to the best of our knowledge, there are not many comparison experiments in this area. Therefore this topic leaves room for more research and represents an open issue. We will address it in this thesis.

Another important topic is the test selection. The interactions of many specifications are parameterized with variables which can take many values. Considering all the possible parameter values will lead to an explosion of the specification interactions. This is only one aspect for which a test generation algorithm can, in principle, generate a large, even infinite number of test cases. Because test execution is limited to a finite number of tests, the test selection is very important. The limitation occurs to be due to time constraints and limitation of resources. We will address the topic of test selection in this thesis.

Now, while we presented the open issues which are addressed in this thesis, it is time to describe the goals which we wanted to achieve and the limitations adopted when researching these open issues. For example, for the tool comparison issue, a limitation is to restrict the set of compared tools. An example of a goal is: we want to apply the selected tools to a case study and to classify them according to the criterion of how many errors of the IUTs each tool discovered. As one can see from these examples, there are limitations and goals because these open issues mean vast domains of research for solving them. We can not claim that we have the final solution for each of them. We needed to restrict ourselves to some specific sub-domains and to fix goals and to try to reach them through this research. When presenting the research, we will describe the limitations and the goals in detail. Now let us outline them.

One topic which proved interesting was the benchmarking of existing tools in this area. One goal of the CdR team was to study the detection power of four known tools Autolink, TGV, TorX and Phact on the Conference Protocol case study. We found it useful also to classify these tools theoretically, by looking at the detection power of the implementation relation on which they are based. This was another goal for us which we tried to reach through this research. To our surprise we could not find an implementation relation for the well-known Autolink tool provided by the Telelogic/TAU Company. Autolink is a tool for test generation which has a user friendly interface. It supports graphical formats which make it easy to represent specifications, test purposes and test cases. The use of test purposes, which are constructed by testers, links the human intelligence to the testing automation. This link contributes in many situations to a good control of the test generation process through the guidance provided by humans. The ITU-T Z.500 supplementary standard requires that any test generated by a test generation algorithm should be sound, which implies the existence of an implementation relation. This requirement was not fulfilled by Autolink, because no implementation relation existed for it. Taking into account all these observations, we wanted to consolidate the conformance foundation of Autolink by building an implementation relation for it. This represented another goal for us in this area.

In the test selection area, one can distinguish the *static* and *dynamic test selection*. For explaining what these terms mean, we should present in more detail the test generation process. In the traditional way, the generation was done in a *batch-oriented* style, which means that the tests were generated or written by human developers and after that executed against the IUT. In other words, generation and execution are two distinct and completely separated phases. Among the tools which work batch-oriented we can mention the commercial tool Autolink and the industrial tool Phact. There is a newer way called *on-the-fly*. This means that the tests are generated and executed against the IUT at the same time. The feedback given by the test execution is used for building up the rest of the test. The prototype tool TorX is working on-the-fly. The static and dynamic selections are related to the on-the-fly style. In the static selection, one selects from the huge set of tests a subset before doing any test generation and execution on-the-fly. This can be done by limiting the parameter ranges to a small number of values. The dynamic selection controls the generation, making the selection at the time of the test generation and execution. In this thesis we will refer to the static selection as *selection* and to the dynamic selection as *control*. In the topic of selection we wanted to formalize relevant heuristics for test selection. When applying test selection heuristics, a reduced set of tests is chosen. For expressing the detection power of the reduced set of tests, we wanted to build up a suitable coverage measure for it. These were the main goals for us in the selection area. In the topic of control we wished to develop options for improving the performance of the TorX tool, the prototype test generation tool of the CdR project and to build a suitable coverage measure for expressing the detection power of a test suite generated on-the-fly. For reaching these goals we mainly concentrated on the stochastic nature of the on-the-fly test generation process.

1.4 The structure of the thesis

While in the previous sections we tried to outline the context and the research goals which are addressed in this thesis, now it is time to introduce the content of the thesis.

- Chapter 2 presents briefly the *ioco* theory of test derivation, which is a prerequisite for all chapters presented in this thesis. Also we provide a summary of the architecture and the main components of TorX, the prototype tool of the CdR project.

- Chapter 3 presents a comparison of four algorithms for test derivation: TorX, TGV, Autolink and UIO algorithms. The algorithms are classified according to the detection power of their conformance relations. Since Autolink does not have an explicit conformance relation, a conformance relation is reconstructed for it. In this way the research presented here strengthens the conformance foundation of Autolink. This work was presented at [Gog01].

- Chapter 4 presents the benchmarking experiment with the four tools TorX, TGV, Autolink and Phact on the Conference Protocol Case Study. We will concentrate especially on the experiment with Autolink on the Conference Protocol because this was our main contribution in the joint effort of benchmarking. The findings were described in [BFdV^{+}99].

- Chapter 5 presents a control technique for on-the-fly test generation through the extension of the TorX algorithm with explicit probabilities. Using these probabilities, the generated test suite can be tuned and optimized with respect to the chances of finding errors in the implementation. This extension was presented in [FGM00].

- Chapter 6 extends the theoretical work from Chapter 5, by presenting experimental results obtained with the probabilistic TorX. The experiment with the Conference Protocol case study confirms that the extension of the algorithm with explicit probabilities which control the test generation leads to improvements in the tests generated with respect to the chances of finding errors in the implementation. The experiment presented here is based on the theory from [FGM00] and uses the context described in [BFdV^{+}99]. This work was presented at [Gog03a].

- Chapter 7 extends the work presented in Chapter 5 into another direction by describing a way to compute the coverage for an *on-the-fly* test generation algorithm based on a probabilistic approach. The *on-the-fly* test generation and execution process and the development process of an implementation from a specification are viewed as stochastic processes. The probabilities of the stochastic processes are integrated in a generalized definition of coverage which can be used for expressing the detection power of a generated test suite. The generalized formulas are instantiated for the *ioco* theory and for the specification of the TorX test generation algorithm. The example which is worked out is based on the theory from [FGM00]. The findings presented in this chapter were presented in [Gog03b].

- Chapter 8 deals with test selection. Since exhaustive testing is in general impossible, an important step in the testing process is the development of a carefully selected test suite. Selection of test cases is not a trivial task. We propose to base the selection process on a well-defined strategy. For this purpose, we formulate two heuristic principles: *the reduction heuristic* and *the cycling heuristic*. The first assumes that few outgoing transitions of a state show essentially different behaviour. The second assumes that the probability to detect erroneous behaviour in a loop decreases after each correct execution of the loop behaviour. We formalize these heuristic principles and we define a coverage function which serves as a measure for the error-detecting capability of a test suite. For this purpose we introduce the notion of a *marked trace* and a distance function on such marked traces. This work was presented at [FGMT02].

- Chapter 9 presents an implementation of the test selection theory described in the previous chapter. Based on a phone specification, an example is worked out in this chapter.

- Chapter 10 gives the conclusions.

Chapter 2

The *ioco* theory for test derivation

The *ioco* (Input Output COnformance) theory deals with conformance testing and test derivation. It is a known theory in this area which serves as a formal basis for several test derivation tools. Among these, we can mention the TorX prototype tool developed within the CdR project. Another important test derivation tool which uses the *ioco* theory as its formal basis, is TGV, developed at IRISA/INRIA Rennes and Verimag Grenoble. We will present more about TGV in Chapter 3.

The *ioco* theory is the theory on which we build the theory in the subsequent chapters. Moreover, we are using the TorX tool in our experiments. The TorX tool is the material realization of the test derivation algorithm developed within *ioco*. Therefore understanding TorX requires an understanding of the *ioco* theory.

Taking into account all these things, we decided to present in an initial chapter the *ioco* theory as a prerequisite of the thesis. This we will do in Section 2.1. In Section 2.2 we will present some aspects regarding how TorX implements the *ioco* theory. This includes the architecture of the tool and some of its component modules.

2.1 The *ioco* theory

In this section we will not present all ingredients of the *ioco* theory. We will extract from this theory those elements which are relevant for the work presented in this thesis. For a full description of the *ioco* theory see [Tre96].

As explained in Chapter 1, the ITU-T Z.500 supplementary standard assumes that a specification is written in a given formalism and that any IUT is modelled by an element m_{IUT} in a given formalism. In the *ioco* theory the formalism used for a specification and for modeling an IUT is the labelled transition systems formalism. An IUT is seen as a black box which exhibits behaviour and interacts with its environment. Being a black box, the IUT can be replaced by its formal model which is only assumed to exist but is not known a priori. For this reason, whenever the *ioco* formalism is mentioning an implementation it means the formal model of it.

Next, the technical details of the *ioco* theory are given. A labelled transition system is defined as follows.

Definition 2.1.1 A labelled transition system is a quadruple $\langle S, L, \rightarrow, s_0\rangle$, where:

1. S is a (countable) non empty set of states;

2. L is a (countable) non empty set of observable labelled actions;

3. $\rightarrow \subseteq S \times (L \cup \{\tau\}) \times S$ is a set of transitions;

4. $s_0 \in S$ is the initial state.

The universe of labelled transition systems over L is denoted by $\mathcal{LTS}(L)$. A labelled transition system is represented in a standard way as a graph or as a process-algebraic behaviour expression. We will use the word automaton for a labelled transition system.

The power set of a set L is the set of all subsets of L, denoted by $\mathcal{P}(L) = \{L' \mid L' \subseteq L\}$. The set of all finite sequences of actions over L is denoted by L^*. The special action $\tau \notin L$ denotes an unobservable action. A trace σ is a finite sequence of observable actions ($\sigma \in L^*$) and $\Rightarrow$ means the observable transition between states ($s \overset{\sigma}{\Rightarrow} s'$ indicates that s' can be reached from state s after performing the actions from trace σ). The empty trace is denoted by ϵ.

Definition 2.1.2 Consider a labelled transition system $p = \langle S, L, \rightarrow, s_0 \rangle$ with $s, s' \in S$ and let $a_i \in L, i \in \mathbf{N}$.

$$\begin{array}{lll} s \overset{\epsilon}{\Rightarrow} s' & =_{\text{def}} & s = s' \text{ or } s \overset{\tau \ldots \tau}{\rightarrow} s'; \\ s \overset{a}{\Rightarrow} s' & =_{\text{def}} & \exists s_1, s_2 : s \overset{\epsilon}{\Rightarrow} s_1 \overset{a}{\rightarrow} s_2 \overset{\epsilon}{\Rightarrow} s'; \\ s \overset{a_1 \ldots a_n}{\Rightarrow} s' & =_{\text{def}} & \exists s_0 \ldots s_n : s = s_0 \overset{a_1}{\Rightarrow} s_1 \overset{a_2}{\Rightarrow} \ldots \overset{a_n}{\Rightarrow} s_n = s'. \end{array}$$

In some cases the transition system will not be distinguished from its initial state. Furthermore we will use $s \overset{a}{\rightarrow}$ (or $s \overset{\sigma}{\Rightarrow}$) to denote $\exists s' : s \overset{a}{\rightarrow} s'$ (or $\exists s' : s \overset{\sigma}{\Rightarrow} s'$).

Definition 2.1.3 Consider a labelled transition system $p = \langle S, L, \rightarrow, s_0 \rangle$ and let $s \in S$, $\sigma \in L^*$. Then:

1. $traces(s) =_{\text{def}} \{\sigma \in L^* \mid s \overset{\sigma}{\Rightarrow}\}$ (the set of traces from s);

2. s **after** $\sigma =_{\text{def}} \{s' \in S \mid s \overset{\sigma}{\Rightarrow} s'\}$ (the set of reachable states after $\sigma \in L^*$).

A failure trace is a trace in which both actions and refusals, represented by a set of refused actions, occur. For this, the transition relation $\rightarrow$ is extended with refusal transitions which are self-loop transitions labelled with a set of actions $A \subseteq L$, expressing that all actions in A and the unobservable action τ can be refused ($s \overset{A}{\rightarrow} s' =_{\text{def}} s = s'$ and $\forall a \in A \cup \{\tau\} : s \overset{a}{\not\rightarrow}$). Please note that one requirement for a state for being able to perform a refusal transition is to be *stable* which means $s \overset{\tau}{\not\rightarrow}$. Another observation is that refusal transitions are obtained from the transition relation $\rightarrow$ of a transition system, but they are not part of this transition relation. Analogously, $\Rightarrow$ is extended to $\overset{\varphi}{\Rightarrow}$, with $\varphi \in (L \cup \mathcal{P}(L))^*$.

Definition 2.1.4 Consider a labelled transition system $p = \langle S, L, \rightarrow, s_0 \rangle$ and let $s \in S$. Let τ be the unobservable action. For defining the extension of the transition relation $\Rightarrow$ to $\overset{\varphi}{\Rightarrow}$, with $\varphi \in (L \cup \mathcal{P}(L))^*$, we use induction on the structure of φ. ($a \in L$, $A \subseteq \mathcal{P}(L)$ and $\varphi' \in (L \cup \mathcal{P}(L))^*$):

1. $s \overset{\epsilon}{\Rightarrow} s' =_{\text{def}} s = s'$ or $s \overset{\tau \ldots \tau}{\rightarrow} s'$;

2. $s \overset{\varphi' a}{\Rightarrow} s' =_{\text{def}} \exists s'' \in S : s \overset{\varphi'}{\Rightarrow} s'' \overset{a}{\Rightarrow} s'$;

3. $s \overset{\varphi' A}{\Rightarrow} s' =_{\text{def}} s \overset{\varphi'}{\Rightarrow} s' \overset{A}{\rightarrow} s'$.

We will also use $s \overset{\varphi}{\Rightarrow}$ to denote $\exists s' : s \overset{\varphi}{\Rightarrow} s'$. For example $s \overset{a\{b,c\}d}{\Rightarrow}$ denotes that s can do a transition labelled with a and reach a state where both c and d are refused, but where d can be done, viz. $\exists s' : s \overset{a}{\Rightarrow} s' \overset{\{b,c\}}{\rightarrow} s' \overset{d}{\Rightarrow}$.

Definition 2.1.5 Let $p \in \mathcal{LTS}(L)$; then we define the failure traces of p as follows: $Ftraces(p) =_{\text{def}} \{\varphi \in (L \cup \mathcal{P}(L))^* \mid p \overset{\varphi}{\Rightarrow}\}$.

For modeling implementations, a special type of transition systems, the input-output transition systems, is used. In these systems the set of actions can be partitioned in a set of input actions L_I and a set of output actions L_U.

Definition 2.1.6 An input-output transition system p is a labelled transition system in which the set of actions is partitioned into input actions L_I and output actions L_U such that $L_I \cup L_U = L$, $L_I \cap L_U = \emptyset$, and for which all input actions are always enabled in any reachable state: whenever $p \overset{\sigma}{\Rightarrow} p'$ then $\forall a \in L_I : p' \overset{a}{\Rightarrow}$.

The universe of such systems is denoted by $\mathcal{IOTS}(L_I, L_U) \subseteq \mathcal{LTS}(L_I \cup L_U)$. For modeling the absence of outputs in a state (a quiescent state) a special action δ ($\delta \notin L$) is introduced and the transformation of the automaton in a *suspension automaton* is used. Formally a state s is quiescent, denoted as $\delta(s)$, if $\forall a \in L_U \cup \{\tau\}$, $\nexists s' : s \overset{a}{\rightarrow} s'$.

Definition 2.1.7 Let $p \in \mathcal{LTS}(L)$, where $L = L_I \cup L_U$. Then the *suspension automaton* of p is the labelled transition system $\langle S_\delta, L_\delta, \rightarrow_\delta, q_0\rangle \in \mathcal{LTS}(L_\delta)$, where:

1. $S_\delta =_{\text{def}} \mathcal{P}(S) \setminus \{\emptyset\}$;
2. $L_\delta =_{\text{def}} L \cup \{\delta\}$;
3. $\rightarrow_\delta =_{\text{def}} \{q \overset{a}{\rightarrow}_\delta q' \mid q, q' \in S_\delta, a \in L, q' = \cup_{s \in q}\{s' \in S \mid s \overset{a}{\Rightarrow} s'\}\} \cup \{q \overset{\delta}{\rightarrow}_\delta q' \mid q, q' \in S_\delta, q' = \{s \in q \mid \delta(s)\}\}$;
4. $q_0 =_{\text{def}} \{s' \mid s_0 \overset{\epsilon}{\Rightarrow} s'\}$.

At point 4) of the definition above, the initial state of the suspension automaton is the set of all the states which can be reached from the initial state of p via an internal transition τ plus the initial state of p. We remind that for an internal transition τ, the empty trace ϵ is seen from the environment. In [Tre96] it is proved that $out(p'$ **after** $\sigma) = out(p$ **after** $\sigma)$, where p is an automaton, p' is the suspension automaton of p and σ is a trace of p.

The suspension traces of $p \in \mathcal{LTS}(L)$ (p is a simple automaton, not a suspension one) are: $Straces(p) =_{\text{def}} Ftraces(p) \cap (L \cup \{L_U\})^*$. For L_U occurring in a suspension trace, we write δ.

One of the main ingredients of the *ioco* theory is the conformance relation. Informally, an implementation is a correct implementation with respect to the specification s and the implementation relation **ioco**$_\mathcal{F}$ if for every trace from $\mathcal{F}$, the set of possible outputs the implementation can generate after performing the trace is allowed by the specification.

Definition 2.1.8 Let $i \in \mathcal{IOTS}(L_I, L_U)$, $s \in \mathcal{LTS}(L_I \cup L_U)$ (i and p are simple automata), and $\mathcal{F} \subseteq L_\delta^*$, then i **ioco**$_\mathcal{F}$ $s =_{\text{def}} \forall \sigma \in \mathcal{F} : out(i$ **after** $\sigma) \subseteq out(s$ **after** $\sigma)$, where $out(p$ **after** $\sigma) =_{\text{def}} \bigcup_{s' \in p \textbf{ after } \sigma}(\{x \in L_U \mid s' \overset{x}{\rightarrow}\} \cup \{\delta \mid \delta(s')\})$.

If $\mathcal{F}= traces(s)$ then **ioco**$_{\mathcal{F}}$ is called **ioconf** and if $\mathcal{F}= Straces(s)$ then **ioco**$_{\mathcal{F}}$ is called **ioco**. The correctness of an implementation with respect to a specification is checked by executing test cases. A test case is seen as a finite labelled transition system which has terminal states called either **pass** or **fail**. An intermediate state of the test case should offer either one input or accept the set of all outputs. The set of outputs is extended with the output θ which means the observation of a refusal (detection of the absence of actions). The class of test cases over L_I and L_U is denoted as $\mathcal{TEST}(L_I, L_U)$. A test suite is a set of test cases. When running a test case against an implementation the test case can give a **pass** verdict or a **fail** verdict. Special care should be taken for the special output δ. A θ-transition can give a **pass** verdict if quiescence is allowed (δ is contained in the output set produced by the specification at that point) or a **fail** verdict if the specification does not allow quiescence at that point. For defining formally a test case run, first, a parallel synchronization operator $\rceil|$ should be introduced. This operator models the communication between a process with θ-transitions and a normal process, i.e., a transition system without θ-transitions.

Definition 2.1.9 The operator $\rceil| : \mathcal{LTS}(L_\theta) \times \mathcal{LTS}(L) \rightarrow \mathcal{LTS}(L_\theta)$ is defined by the following inference rules ($L_\theta = L \cup \{\theta\}$):

1. $u \xrightarrow{\tau} u' \quad \vdash \quad u\rceil|p \xrightarrow{\tau} u'\rceil|p;$
2. $p \xrightarrow{\tau} p' \quad \vdash \quad u\rceil|p \xrightarrow{\tau} u\rceil|p';$
3. $u \xrightarrow{a} u',\ p \xrightarrow{a} p',\ a \in L \quad \vdash \quad u\rceil|p \xrightarrow{a} u'\rceil|p';$
4. $u \xrightarrow{\theta} u',\ p \overset{\tau}{\not\rightarrow}, \forall a \in L, u \overset{a}{\not\rightarrow}$ or $p \overset{a}{\not\rightarrow} \quad \vdash \quad u\rceil|p \xrightarrow{\theta} u'\rceil|p.$

Using this operator the test run can be defined formally in the following way.

Definition 2.1.10 Let $t \in \mathcal{TEST}(L_I, L_U)$ be a test case and $i \in \mathcal{IOTS}(L_I, L_U)$ be an implementation. A test run of t and i is a trace of the synchronous parallel composition $t\rceil|i$ leading to a terminal state of t: σ is a test run of t and i $=_{\text{def}} \exists i' : t\rceil|i \overset{\sigma}{\Rightarrow}$ **pass**$\rceil|i'$ or $t\rceil|i \overset{\sigma}{\Rightarrow}$ **fail**$\rceil|i'$.

In the definition above, please note that $\mathcal{TEST}(L_I, L_U) \subseteq \mathcal{LTS}(L_\theta)$ and that $\mathcal{IOTS}(L_I, L_U) \subseteq \mathcal{LTS}(L)$. An implementation i passes a test case t if all their test runs lead to the **pass**-state of t. The implementation passes a test suite T if it passes all test cases in T. If i does not pass the test suite, it is said to fail. The conformance relation used between an implementation i and a specification s is **ioco**$_{\mathcal{F}}$. In the ideal case, the implementation should pass the test suite if and only if the implementation conforms. In this case the test suite is called complete. In practice because the test suite can be very large we have to restrict to test suites that can only detect non-conformance, but cannot assure conformance. Such test suites are called sound. Exhaustiveness of a test suite means that the test suite can assure conformance (but it can also reject conforming implementation). For deriving tests the following specification of an algorithm is presented in [Tre96]:

The specification of the *ioco* test derivation algorithm

Let S be the suspension automaton of a specification and let $\mathcal{F} \subseteq traces(S)$; then a test case $t \in \mathcal{TEST}(L_I, L_U)$ is obtained by a finite number of recursive applications of one of the following three nondeterministic choices:

1. (*terminate the test case*)

   $t =$ **pass**

2. (*supply an input for the implementation*)

Take $a \in L_I$ such that $\mathcal{F}_a \neq \emptyset$

$t = a\,; t_a$

where $\mathcal{F}_a = \{\sigma \mid a\,; \sigma \in \mathcal{F}\}$, $S \xrightarrow{a}_\delta S_a$ and t_a is obtained by recursively applying the algorithm for S_a and $\mathcal{F}_a$;

3. (*check the next output of the implementation*)

$$\begin{aligned} t = \; & \sum\{x\,; \mathbf{fail} \mid x \in L_U \cup \{\theta\}, \overline{x} \notin out(S), \epsilon \in \mathcal{F}\} \\ & + \sum\{x\,; \mathbf{pass} \mid x \in L_U \cup \{\theta\}, \overline{x} \notin out(S), \epsilon \notin \mathcal{F}\} \\ & + \sum\{x\,; t_x \mid x \in L_U \cup \{\theta\}, \overline{x} \in out(S)\} \end{aligned}$$

where $\overline{x}$ is a notation for x in which the δ action is replaced by a θ action and vice versa, $\mathcal{F}_x = \{\sigma \mid \overline{x}\,; \sigma \in \mathcal{F}\}$, $S \xrightarrow{\overline{x}}_\delta S_x$ and t_x is obtained by recursively applying the algorithm for S_x and $\mathcal{F}_x$. The summation symbol $\sum$ and the plus symbol $+$ mean choice and the symbol ; means concatenation.

The algorithm has three choices. In every moment it can choose to supply an input a from the set of inputs L_I or to observe all the outputs ($L_U \cup \{\theta\}$) or to finish. When it finishes, because this does not mean that the algorithm detected an error, it finishes with a **pass** verdict. After supplying an input, the input becomes part of the test case and the algorithm is applied recursively for building the test case. When it checks the outputs, if the current output is present in $out(S)$, that output will become also part of the test case and the algorithm will be applied recursively. If the output is not present in $out(S)$ the algorithm finishes with a **fail** verdict if the empty trace is considered an element of $\mathcal{F}$. If the empty trace is not in $\mathcal{F}$ then the verdict will be **pass**.

This algorithm satisfies the following properties (for a proof see [Tre96]):

Theorem 2.1.11 *(Completeness of the ioco test derivation algorithm)*

1. *A test case obtained with this algorithm is sound with respect to* $\mathbf{ioco}_{\mathcal{F}}$.

2. *The set of all possible test cases that can be obtained with the algorithm is exhaustive.*

For a good understanding of the algorithm let us apply it to the suspension automaton for a candy machine from the right-hand side of Figure 2.1. On the left-hand side of the figure the simple automaton of the candy machine is represented. The suspension automaton is obtained from the automaton from the left-hand side of the figure according to Definition 2.1.7. For simplifying the presentation, the transformation from the simple automaton to a suspension one is not discussed here and some of the interactions between a potential user of candy and the machine itself are removed, as for example the interaction of inserting a coin. The label set of this suspension automaton which we will denote as candy is the union of the set of inputs $L_I = \{but\}$ and of the set of outputs $L_U = \{liq, choc, \delta\}$. We recall that for a suspension automaton the set of outputs is extended with the null output δ. After pushing the button but, the machine will produce liquorice (liq) or nothing (δ). In state 2, when the button but is pushed again the candy machine will produce liquorice or chocolate ($choc$). If nothing was produced (the machine is now in state 4) and the button is pressed, the machine will provide only chocolate. After the chocolate or the liquorice is given, pushing the button will give no response (δ output).

The implementation of this algorithm in the TorX architecture usually generates the test cases on-the-fly. To simplify the explanation we will use a batch-oriented approach. The set $\mathcal{F}$ equals the

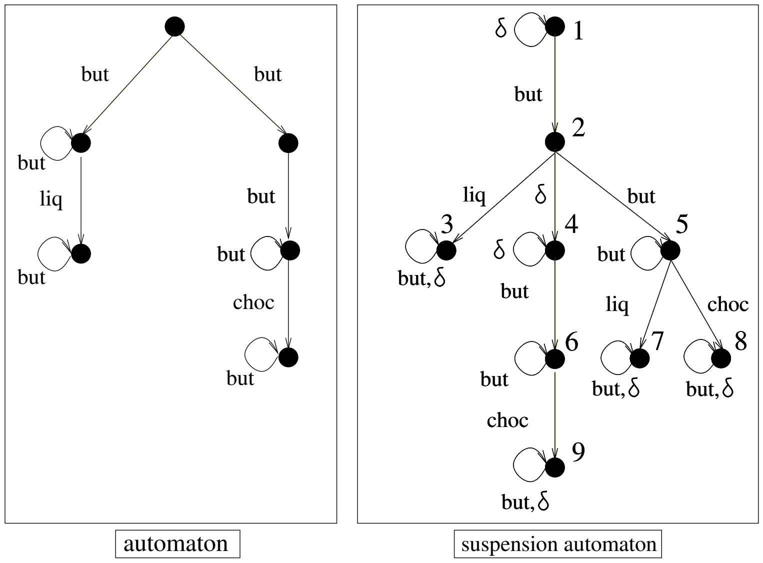


Figure 2.1: The specification for a candy machine.

set *traces*(candy). In the execution sequence, by S_1, ..., S_9 we mean the states of the candy machine labelled with 1, ..., 9 in Figure 2.1.

A possible execution of the algorithm is:

- First *Choice 2* (*select an input*) ($S = S_1, \mathcal{F} = traces(S_1)$):
  $t = but; t_1$;
- To obtain t_1 the algorithm chooses *Choice 2* ($S = S_2, \mathcal{F} = traces(S_2)$):
  $t_1 = but; t_2$;
- Now *Choice 3* is selected (*check the output*) for computing t_2 ($S = S_5, \mathcal{F} = traces(S_5)$, $\epsilon \in \mathcal{F}$):
  $t_2 = liq; t_{21} + choc; t_{22} + \theta;$ **fail**;
- For *liq* the algorithm finishes (*Choice 1*) ($S = S_7, \mathcal{F} = traces(S_7)$):
  $t_{21} =$ **pass**;
- For *choc* the algorithm again checks the output (*Choice 3*):
  $t_{22} = liq;$ **fail** $+ choc;$ **fail** $+ \theta; t_{31}$ ($S = S_8, \mathcal{F} = traces(S_8), \epsilon \in \mathcal{F}$);
- If θ is observed, it chooses *Choice 1* ($S = S_8, \mathcal{F} = traces(S_8)$):
  $t_{31} =$ **pass**.

The resulting test case is shown in Figure 2.2. We remind that the output θ means the observation of a refusal. We see that *but but liq* is correct behaviour. We can also see that *but but choc choc* is incorrect behaviour.

2.2 The TorX prototype tool

In this section we will present how TorX is implementing the *ioco* test derivation algorithm described in Section 2.1. We will give the TorX design ([BFdV+99]) and we will explain the component parts

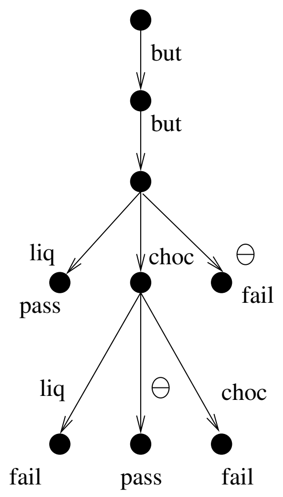


Figure 2.2: The test case generated for a candy machine.

which form the tool. We will give a high-level description of TorX. The high-level description of TorX gives the reader the possibility to understand the TorX architecture without entering too much in its technical details.

The main architecture of TorX is given in Figure 2.3. As one can see the building modules of TorX are named: explorer, primer, driver and adapter. In the figure, one can see also that TorX can be used on-the-fly, by using all these components, or batch-oriented, by the use of the driver and adapter modules. Designing the tool in a modular form makes TorX really flexible. The other characteristic attribute of TorX, the openness, is given by the fact that TorX supports the integration with a third party tool. This is because, whenever possible, it uses industrial standard interfaces to link its modules. One example is the GCI interface (Generic Compiler Interpreter [BB96]) which connects the adapter and SUT. When no standard interface is used, TorX connects components by pipes over which textual commands and responses are exchanged. Now let us outline the functions of each module.

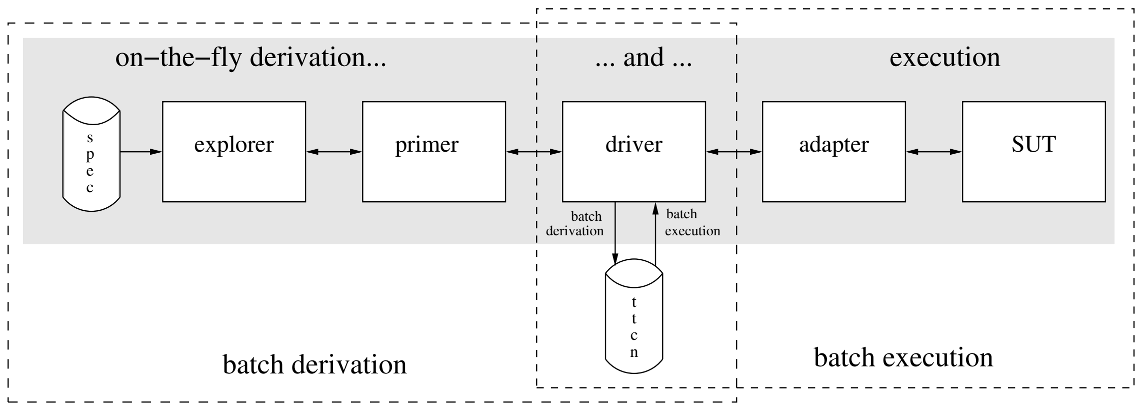


Figure 2.3: The TorX prototype tool.

The explorer module is in charge of the exploration of the specification states. It can give for a state the set of actions which are allowed in that state. These functionalities of the explorer are used by the primer for the test derivation. The explorer is specification-language dependent. TorX is using existing tools for implementing the explorer. Currently it uses CAESAR ([CAESAR]) as its LOTOS explorer and SPIN ([SPIN]) as its PROMELA explorer. Possible future extensions of TorX include the use of other specification languages, the semantics of which can be expressed in labelled trasition systems such as SDL and, consequently, the use of other existing tools for the explorer.

The primer is in charge of the test derivation. It uses the functionalities provided by the explorer for implementing the 3 choices of the *ioco* test derivation algorithm presented in Section 2.1: (*terminate the test case*), (*supply an input for the implementation*) and (*check the next output of the implementation*). The random selection of the choices is made by the driver.

Initially the driver was using random number generators for implementing the nondeterminism of the *ioco* test derivation algorithm. Now it can also use, for implementing the nondeterminism, probabilities assigned to the three choices or to partitions of inputs. The driver can store the tests in a storage (indicated with TTCN in the figure) for using them later for batch-oriented execution. For the on-the-fly execution the test events (such as sending an input to SUT) are obtained directly from the primer. The driver uses the adapter for sending the inputs to the SUT or for checking the outputs of the SUT.

The adapter makes the connection between the rest of modules and SUT by sending inputs to it and receiving outputs from it. It is also responsible for the mapping between time-outs and the quiescence action δ.

These are the main modules which form TorX. In the final version of TorX, at the end of the CdR project, this architecture could be changed and expanded. This is because the theory and the experiments done within the CdR project could improve its design further by adding or transforming the modules. Because we use TorX in the experiments described in this thesis in the form which we presented in this section, it is out of our scope to give all the changes and extensions which TorX will have in time.

The presentation of the *ioco* theory and TorX gives the technical background for the things presented in this thesis. The next chapter describes the comparison between existing tools for test derivation, including TorX.

Chapter 3

A comparison of TorX, Autolink, TGV and UIO Test Algorithms

3.1 Introduction

There are several tools for test generation using different algorithms. In the introductory chapter we mentioned some names such as TorX or Autolink without going into details. Because each tool is an implementation of an algorithm for test derivation in this chapter we will denote with the name of a tool both the tool itself and the algorithm on which the tool is based. For example, when saying TorX we will understand the tool, as it is described in Section 2.2, and the algorithm presented in Section 2.1, which is the algorithm implemented by TorX. This convention is used only in this chapter and does not apply to the other chapters of the thesis.

From a user point of view, for selecting one tool for use, it is interesting to know what the possibilities and limits of these algorithms are. Work in this direction was done within the CdR project. The first year of the CdR project was devoted to a comparison between the performance of TorX, the test generation algorithm of the project, and the performance of two other algorithms: Autolink, which is part of the commercial tool TAU/Telelogic [SEG$^+$98] and Phact, which is an UIO ([ADLU91]) algorithm used within Philips [FMMvW98]. The results were presented in [BFdV$^+$99], [BRS$^+$00]. Because the benchmarking was received with interest by the scientific world, it was extended [HFT00] the following year by including another algorithm in the experiments: TGV developed at IRISA/INRIA Rennes and Verimag Grenoble ([FJJT96]).

The comparison was done by means of practical experiments. In addition we felt it would be useful to complement the experiment by classifying the four algorithms from a theoretical point of view. The findings of our work were presented at [Gog01]. For researchers who are acquainted with the different approaches to automated test derivation, the results of the research presented here, may seem straightforward. However, due to the fact that the different schools use different notations, we think that the mere act of expressing these methods in the same framework is already worthwhile. Another interesting finding of our research is that Autolink does not have an explicit conformance relation. So our work, in which a conformance relation is reconstructed for this algorithm, consolidates its conformance foundation. The theory presented here is quite general because it treats a set of test generation algorithms which are well known and which come from a wide range of domains: academia, commercial and industry.

In our comparison, the error detection power of the algorithms is judged as depending on the conformance relation which they implement. Because two algorithms from four, TGV and TorX, use the *ioco* theory as their formal foundation, we expressed the conformance relations of all algorithms in terms of *ioco* theory (which can be done because it is the most general one).

The algorithms are judged in the limit case when they exhaustively generate a large, even infinite number of tests. Even in the limit, they can detect only the erroneous implementations which their conformance relation can detect.

As explained above, in this chapter we will relate different conformance relations. Care should be paid to make it explicit on which domains these conformance relations are well-defined, and for which domains of models the comparisons hold. Because, apart from the differences between the relations themselves, the difference is also determined by the classes/domains of models for which they are defined. We will extend this discussion throughout this chapter.

The theoretical comparison is presented in this chapter in the following way. The *ioco* theory and TorX were depicted in Chapter 2. For this reason it is not necessary to have a separate section in this chapter for describing them. The rest of the algorithms is presented in a number of sections in the following order: TGV in Section 3.2, UIO in Section 3.3, and Autolink in Section 3.4. Section 3.5 describes the conclusions.

3.2 TGV

TGV ([JM99]) is a tool for automatic test derivation developed at IRISA/INRIA Rennes and Verimag Grenoble ([FJJT96]). We will outline some elements of TGV below. For a precise description of TGV see [JM99] and [FJJT96]. TGV is connected to several simulators: the SDL Simulator of ObjectGeode from Verilog ([ALHH93]) and the Lotos Simulator from CADP ([FGK$^+$96]). The inputs of TGV are a state graph produced by a simulator and an automaton formalizing the behavioural part of a test purpose. The test purposes can be generated automatically or by hand. The output is a test case which is produced in the standard TTCN format ([ISO92]). The algorithm was used batch-oriented for testing a variety of IUTs, in industrial and military application domains ([JM99]).

The tool TGV is based on the **ioco** conformance relation. The authors of TGV prove that TGV is sound and exhaustive with respect to this conformance relation ([JM99, FJJT96]).

Because TorX and TGV are both exhaustive with respect to **ioco** this means that from a theoretical point of view TGV, and TorX in the limit case when they exhaustively generate a large, even infinite number of tests have the same error detection power (the **ioco** detection power).

3.3 UIO and UIOv algorithms (Phact)

One of the early methods used for automatic test derivation was the UIO ([ADLU91]) – Unique Input/Output sequence – technique. Later this method was developed in a new methodology (called UIOv – [VCI90]) more powerful in the detection of the IUTs which do not conform to a specification. The UIO methodology is still used today. One example is Phact (*PHilips Automated Conformance Tester* [FMMvW98], [MRS$^+$97]) – a tool developed at the Philips Research Laboratories in Eindhoven for deriving test cases in a TTCN format and executing them. The heart of the tool is the Conformance Kit ([vdBKKW89]) developed by KPN.

Although [VCI90] shows that the UIO relation is not complete several tools from industry (such as the Conformance Kit) implement the UIO method in place of UIOv, due to the complexity of the UIOv method. In our comparison between the four algorithms, we will say some words about the UIOv algorithm. As we did up to this point, we will judge the detection power of Phact by looking at the detection power of its conformance relation: UIO. This will be realized by going through the following steps: Subsection 3.3.1 will give a survey of the UIO and UIOv theory; Subsection 3.3.2 will present the translation of the UIO and UIOv concepts to *ioco* concepts and the definition of the $\mathbf{iocofsm}_{\mathcal{F}}$ conformance relation. Roughly speaking, $\mathbf{iocofsm}_{\mathcal{F}}$ is a restriction of the original $\mathbf{ioco}_{\mathcal{F}}$ defined for LTSs which are the image of a deterministic, connected and minimal FSM (finite-state-machine). Subsection 3.3.3 will give the classification of the **UIO**, **UIOv** and **iocofsm** conformance relations.

3.3.1 A survey of the UIO and UIOv theory

The UIO and UIOv methods apply to deterministic FSM models (finite-state-machine) which are commonly referred to as Mealy machines ([Koh78]).

Definition 3.3.1 An FSM is a quintuple $M = \langle S, I, O, NS, Z\rangle$ where:

1. S is a finite set of states, the typical elements of which are denoted $s_0, ..., s_n$;
2. I is a finite set of inputs, the typical elements of which are denoted $i_1, ..., i_m$;
3. O is a finite set of outputs, the typical elements of which are denoted $o_1, ..., o_r$ $(O \cap I = \emptyset)$;
4. NS (next state) is the transfer function of type $S \times I \rightarrow S$;
5. Z is the output function of type $S \times I \rightarrow O$.

Here it is understood that NS and Z are total functions. Moreover the FSM has: 1) an initial state $s_0 \in S$; 2) a reset transition $r \in I$ which assures that from every state it is possible to return to the initial state – for every state $s \in S$ we have that $NS(s, r) = s_0$; 3) a *null* $\in O$ output produced by some inputs in some states. Every transition of an FSM is labeled with a pair input/output of signals (a signal is either an input or an output).

One thing deserves attention, namely the presence of the null output in the set of outputs. The most natural translation of the *null* output in *ioco* terms is to equal it to δ. But there is a difference between the approaches of the FSM view compared to the *ioco* view. In the FSM view, *null* (or δ) is part of the output set while in the *ioco* view δ is not part of the output set, it is a special output added to it. From a user point of view, the difference between the two views means the following. In the FSM view, the user will mark the points (states of the specification) where the checking of the absence of the outputs is wanted (or needed). This means extra work. In the *ioco* view the states in which no output is produced (or δ is present) is automatically determined. We will come back to this discussion in Section 3.3.2.

The reflexive, transitive closure of the transfer function is defined as in the following.

Definition 3.3.2 Let $M = \langle S, I, O, NS, Z\rangle$ be an FSM. Let $s \in S$ be a state, ϵ the null (empty) sequence, $\sigma \in I^*$ a sequence of inputs and $i \in I$ an input. Then the closure $NS^* : S \times I^* \rightarrow S$ of the transfer function is:

1. $NS^*(s, \epsilon) = s$;
2. $NS^*(s, \sigma i) = NS(NS^*(s, \sigma), i)$.

The language of the FSM is $\mathcal{L}(M) = \{i_1/o_1...i_k/o_k \in (I \times O)^* \mid k \in \mathbf{N}, \forall j \in \mathbf{N}, 1 \leq j \leq k : Z(NS^*(s_0, i_1...i_{j-1}), i_j) = o_j\}$. In the formula above, $i_1/o_1...i_k/o_k$ means a sequence of input/output transitions. An FSM is minimal if there does not exist another FSM which has the same language and which has a smaller number of states. An FSM is connected if from every state s of the FSM it is possible to reach every other state t via a sequence of inputs.

Definition 3.3.3 An FSM $M = \langle S, I, O, NS, Z\rangle$ is connected iff
$\forall s, t \in S, \exists \sigma \in I^* : NS^*(s, \sigma) = t.$

A transition of an FSM is a quadruple $\langle s, i, o, t\rangle$ with $s, t \in S$ and $i \in I$ and $o \in O$ such that $NS(s, i) = t$ and $Z(s, i) = o$. We write $s \xrightarrow{i/o} t$ for this transition. A *UInOut* sequence – Unique Input Output – for a state s, roughly speaking, is an I/O behaviour not exhibited by any other state. In the remainder of this chapter, we restrict the discussion to FSMs which are connected and minimal and which have a *UInOut*-sequence for every state (not any connected and minimal FSM has a *UInOut* for every state [LY96]). In general there are more *UInOut*s for a given state. In such a case we can choose one of them. In the remainder of this chapter by *UInOut*(s) we mean the chosen *UInOut* for a state s.

Example Let us consider the FSM of the specification *spec* from Figure 3.1. The initial state of this FSM is the state 1. For this FSM, the set of inputs is $I = \{a, d, r\}$ and the set of outputs is $O = \{b, c, e, null\}$. The reset signal r which is followed by the *null* output is present in every state. For simplification it was not represented in the figure. Examples of *UInOut* sequences of the states of the specification are given in Figure 3.2 (a). The state 1 has a distinguishing input/output pair a/b which is its *UInOut*(1); the *UInOut* for the states 2 and 3 are formed by concatenating two transitions: *UInOut*(2) is $a/c.a/b$ and *UInOut*(3) is $d/e.a/b$.

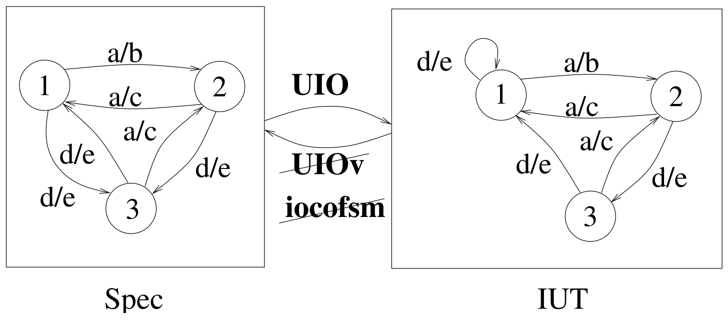


Figure 3.1: The difference between **UIO**, **UIOv** and **iocofsm** conformance relations.

For applying the UIO and UIOv methods it is necessary that: 1) the FSM of the specification is connected, 2) it is minimal and 3) the number of the states of the implementation (FSM_i) is less than or equal to the number of the states of the specification (FSM_s).

For the UIO and UIOv algorithms the approach taken for checking the correctness of an implementation FSM_i to its specification FSM_s is to test that every transition of the specification is correctly implemented by FSM_i. The procedure for testing a specified transition from state s to state t (of the specification FSM_s) with input/output pair i/o consists of three steps:

1. FSM_i is brought to state s;

2. input i is applied to FSM_i and the output produced is checked for whether it is o;

3. the state FSM_i reached after the application of input i is checked for whether it is t.

These three steps are only an abstract procedure. There are several ways, for example, to perform the check of step 3) (although each way is checking input/output behaviours, not directly inspecting the state). We turn this into a more concrete procedure (organized on three sequences to be executed in succession), which we call **TestCase**. Regarding the parameters of this procedure, we will give some explanations below.

The step 3) of the procedure **TestCase** is implemented in a different way by an UIO algorithm than by an UIOv algorithm. So we will introduce the type of the algorithm as a parameter of the procedure (**TestCase**(UIO,...) or **TestCase**(UIOv,...)). This parameter we will denote as u and it can take one of the two values UIO or UIOv (for example, it can be a charstring, 'UIO' or 'UIOv'). For building a test, the knowledge of the *UInOuts* of the states is used. So the set of pairs ⟨state, *UInOut*(state)⟩, which we will denote as *SUIO*, will be another parameter of the procedure. Although FSM_i is used within the procedure, the building of the test does not depend on it; so we will not put FSM_i in the list of parameters.

TestCase(u, *SUIO*, FSM_s, $s \xrightarrow{i/o} t$)

1. a sequence σ which brings FSM_i from its current state s_0 to state s (we call this a transferring sequence);

2. input i to check the output o;

3. a sequence to verify that FSM_i reached t (we explain this below).

The step 3) of the procedure makes the difference between the UIO and UIOv algorithms. The UIO algorithm implements it by simply checking that the implementation exhibits the *UInOut* of that state (t). Using the UIO algorithm, some faults may go undetected, however ([VCI90]). The problem of the UIO algorithm is that although each UIO sequence is unique in the specification, this uniqueness need not hold in a faulty implementation (see the example at the end of this subsection). The solution proposed by the UIOv algorithm is: 1) to check that the implementation exhibits the *UInOut* behaviour of that state (t); 2) it does not exhibit the *UInOut* behaviour of any other state k. Because UIO is simpler than UIOv, usually the tools implement the UIO method (Phact uses the UIO method).

The authors of UIOv ([VCI90]) show that their method can verify whether the FSM of the IUT is the same as the FSM of the specification, with other words UIOv can detect any missing and erroneous state and I/O of an IUT. This happen under the assumptions on which UIOv is applied: A) the specification FSM is connected, B) the specification FSM is minimal and C) the number of the states of the implementation is less than or equal to the number of the states of the specification.

It is also interesting to observe that, choosing to check each transition of an FSM can be seen as a test selection method (the selection works in the following way: in a state which has many transitions, only the transitions which are not checked yet are considered for the test generation). This shows that one of the central topics of this thesis was treated by two of the first methods for automatic test generation. Because the UIO and UIOv methods have limits (which will be discussed in more detail later), this particular way of doing test selection does not offer the final solution to this topic, although using it many errors in IUTs can be discovered.

Example The specification from Figure 3.1 has three states and the implementation has also three states. The FSM of the specification is connected and minimal. So the requirements for applying the UIO (UIOv) method are fulfilled.

State 1: a/b
State 2: a/c.a/b
State 3: d/e.a/b

a) UInOuts

UInOuts Verification
r/null.a/b
r/null.a/b.a/c.a/b
r/null.a/b.d/e.d/e.a/b

Transitions State 1
r/null.a/b.a/c.a/b
r/null.d/e.d/e.a/b

Transitions State 2
r/null.a/b.a/c.a/b
r/null.a/b.d/e.d/e.a/b

Transitions State 3
r/null.a/b.d/e.a/c.a/c.a/b
r/null.a/b.d/e.d/e.a/b

b) UIO test cases

State 1
~UIO(2): r/null.a/b.a/c
~UIO(3): r/null.d/e.a/c

State 2
~UIO(1): r/null.a/b.a/c
~UIO(3): r/null.a/b.d/e.a/c

State 3
~UIO(1): r/null.a/b.d/e.a/c
~UIO(2): r/null.a/b.d/e.a/c.a/c

c) Specific UIOv test part

Figure 3.2: Test cases derived for UIO (b) and UIOv (b, c) conformance relations.

There is a subtle difference between the IUT and the specification: the transition d/e of the state 1 goes in the specification to the state 3 and in the IUT to the same state 1. The *UInOut*(3) (Figure 3.2 (a)) does not succeed only in the state 3 of the implementation but it succeeds also in the state 1. So applying an UIO algorithm on the set of *UInOut* from Figure 3.2 (a) will let the faulty IUT go undetected, but with an UIOv algorithm it will be detected.

The test derivations using the UIO and UIOv methods are given in Figure 3.2 (b) and (c). The verdict is not shown. It is implicit: if the IUT does not produce the expected output the verdict will be **fail**; if the IUT correctly executes all the test cases the verdict will be **pass**. The tests from the first subpart of Figure 3.2 (b) (the *UInOuts Verification* part) check the *UInOut* of every state in the implementation. Although this part is redundant, we add it to make the difference between an UIO algorithm and an UIOv algorithm more explicit. The next part from Figure 3.2, (b) checks every transition of the specification. We will illustrate the construction of the tests from part (b) by showing it for the test: $r/null.a/b.d/e.a/c.a/c.a/b$. This test is built from four parts: 1) the sequence $r/null$ brings the FSM in the initial state; 2) the sequence $a/b.d/e$ brings the FSM in state 3; 3) the sequence a/c is the input/output pair of the transition which is checked; 4) now a correct implementation is supposed to be in state 2; the sequence $a/c.a/b$ is *UInOut*(2). Please note that the derived set of test cases is not optimal: the sequence $a/b.d/e$ which brings the FSM in state 3 is not the shortest one. The shortest one is d/e. For showing the difference between an UIO algorithm and an UIOv algorithm this aspect does not matter. For this reason, we did not pay attention to optimization criteria when we constructed this example. The specific part of the UIOv algorithm (Figure 3.2 (c)) tests that every *UInOut* is not exhibited by another state of the implementation. From the set of tests from part (c), let us take the test $r/null.a/b.d/e.a/c.a/c$. This test is formed by: 1) the reset sequence $r/null$

which brings the FSM in the initial state; 2) the sequence $a/b.d/e$ which brings the FSM in the state 3; 3) the sequence $a/c.a/c$ which is formed by the sequence of inputs $a/.a/$ (from *UInOut*(2)) and by the sequence of outputs $/c./c$ produced by the specification when $a/.a/$ is applied in the state 3. The set of tests produced by the UIOv algorithm is composed by the union of the set of tests from Figure 3.2 (b) and from Figure 3.2 (c).

3.3.2 The translation of the UIO and UIOv concepts to the *ioco* concepts

For making the translation of the UIO and UIOv concepts to the *ioco* concepts, first we should represent an FSM as a labelled transition system. The conversion can be made easily by adding an intermediate state after each input.

Definition 3.3.4 The labelled transition system $M' = \langle S', L', \longrightarrow, s_0\rangle$ of an FSM $M = \langle S, I, O, NS, Z\rangle$ is defined by:

1. $S' = S \cup (S \times I)$;
2. $L' = I \cup O$;
3. $\longrightarrow \subseteq S' \times L' \times S'$,
   $\longrightarrow = \{s \xrightarrow{i} \langle s, i\rangle, \langle s, i\rangle \xrightarrow{Z(s,i)} NS(s, i) \mid s \in S, i \in I\}$;
4. s_0 is the initial state of M.

By $\langle s, i\rangle \in (S \times I)$ we denote an intermediate state of a labelled transition system of an FSM, which separates the input i from the output $Z(s, i)$.

Example Let us consider the FSM specification from Figure 3.1. The labelled transition system of this specification is given in Figure 3.3.

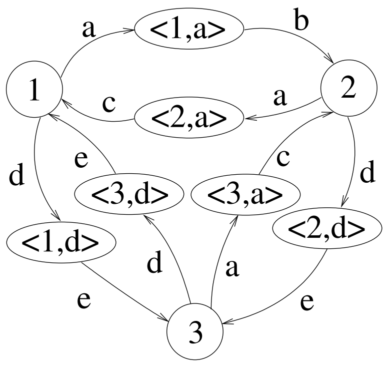


Figure 3.3: The labelled transition system of the specification.

The correspondence between an FSM and its labelled transition system is very simple because the labelled transition system of an FSM has only extra states which separate an input from an output. We will exemplify this for some elements of the FSMs. A sequence $\sigma = i_1/o_1...i_n/o_n$ ($n > 0, i_j \in I, o_j \in O, j = 1, ..., n$) produced by an FSM is translated into a correspondent trace σ' for the labelled transition system of the FSM, with $\sigma' = i_1o_1...i_no_n$. In the same way, we can obtain a correspondent set of traces for the set of tests produced by the procedure **TestCase**. The same thing can be also done for the *UInOut* of the states of an FSM.

In this point one thing deserves attention. We recall from the previous section the discussion regarding the presence or the absence of *null* in the output set. Now the question which arises is whether an LTS derived from an FSM represents a common automaton or a special one, namely a special type of a suspension automaton? The answer differs in function of the view adopted (the FSM view or the *ioco* view). Because the final goal of this section is to relate UIO and UIOv to the (restriction of) **ioco** conformance relation on which TorX and TGV are based (the **ioco** conformance relation works on suspension traces), we will adopt an *ioco* view in this section. In this view we can consider an LTS derived from an FSM as being a special type of a suspension automaton. Because the null output is part of the set of outputs it results that $L' = L_\delta$. These considerations are made only for this section. In the section related to Autolink, Section 3.4, we need to adopt the FSM view for comparing Autolink with UIO and UIOv.

Before going further in translating UIO and UIOv in *ioco* terms, we should also pay attention to the application of the $\mathbf{ioco}_{\mathcal{F}}$ conformance relation on LTSs derived from FSMs. It can be observed that an LTS constructed from an FSM is not input-enabled in the intermediate states, while this is requested for any state of the implementation in the definition of $\mathbf{ioco}_{\mathcal{F}}$ (see Definition 2.1.8). For proceeding further with the comparison, we need to consider only a part named $\mathbf{iocofsm}_{\mathcal{F}}$ of the full $\mathbf{ioco}_{\mathcal{F}}$ defined for LTSs (see Definition 3.3.4) which are the image of a deterministic, connected, and minimal FSM.

Definition 3.3.5 The conformance relation $\mathbf{iocofsm}_{\mathcal{F}}$ is a part of the full $\mathbf{ioco}_{\mathcal{F}}$ on which the following restrictions apply

1. the implementation is an LTS derived from an FSM;

2. the specification is an LTS derived from an FSM;

3. $\mathcal{F}$ consists of traces formed by sequences of I/O pairs which ends in an input, i.e. $\mathcal{F} = \{i_1 o_1 ... i_n o_n i_{n+1} \mid i_j \in I \ \ (1 \leq j \leq n+1), o_j \in O \ \ (1 \leq j \leq n), n \geq 0\}$;

In the remainder of this chapter, we will compare the UIO and UIOv using **iocofsm**, which is $\mathbf{iocofsm} = \mathbf{iocofsm}_{traces'(s)}$, where the specification s is the LTS derived from the FSM of a specification and $traces'(s) \subset traces(s)$ is formed by all the traces of s which finish into inputs. In the remainder of this section we will not distinguish between $traces'(s)$ and $traces(s)$ because for an FSM any trace is formed by sequences of input/output signals and it makes sense to checks the outputs produced by an implementation only for the traces which finish into inputs. Now, if one takes the FSM view, with *null* (or δ) in the set of outputs, then **iocofsm** is part of **ioconf**. For comparing UIO and UIOv with Autolink this view should be adopted (we will come back to this discussion in Section 3.4). If one considers the *ioco* approach, then **iocofsm** is part of **ioco**. Because, in this section, our goal is to relate UIO and UIOv with **ioco**, we will adopt the *ioco* view in this section.

So, in the remainder of this chapter by an FSM we will understand the labelled transition system of an FSM and by S (or sometimes St) the set of states which are not intermediate. Now, we will give a formal definition of the **UIO** and **UIOv** conformance relations. But, before doing this, we will introduce some terminology.

A test produced by the procedure **TestCase** is a trace. So a test suite is a set of traces. An FSM **passes** a test suite if the test suite is contained in the set of traces of the FSM.

Definition 3.3.6 Let w be an FSM and T a test suite. Then:
w **passes** $T \Leftrightarrow T \subseteq traces(w)$.

The Unique Input Output Sequence of a state is defined below.

Definition 3.3.7 Let FSM_s be an FSM of a specification and k a state of FSM_s, k is not an intermediate state.

A trace σ is a *UInOut* of $k \Leftrightarrow k \overset{\sigma}{\Rightarrow} \wedge \forall s', s' \neq k, s' \neq \langle \_, \_ \rangle : s' \overset{\sigma}{\not\Rightarrow}$.

In the definition above, by $s' \neq \langle \_, \_ \rangle$ we mean that s' is not an intermediate state, where _ stands for any. The test suites produced by **TestCase** are defined as in the following.

Definition 3.3.8 Let FSM_s be the FSM of a specification. Let I and St be the input set and state set of FSM_s. Let $SUIO$ be the set formed by the pairs of the states from St and their *UInOut*, that is $SUIO = \{\langle s, UInOut(s) \rangle \mid s \in St\}$. Then,

1. $T_{UIO} = \cup_{i \in I, s, t \in St}\{$ **TestCase**(UIO, $SUIO$, FSM_s, $s \overset{i/Z(s,i)}{\longrightarrow} t$)$\}$ is the test suite formed by the set of test cases produced by the procedure **TestCase** using an UIO algorithm for the step 3) of the procedure;

2. $T_{UIOv} = \cup_{i \in I, s, t \in St}\{$ **TestCase**(UIOv, $SUIO$, FSM_s, $s \overset{i/Z(s,i)}{\longrightarrow} t$)$\}$ is the test suite formed by the set of test cases produced by the procedure **TestCase** using an UIOv algorithm for the step 3) of the procedure.

Now we will formally define the **UIO** and **UIOv** conformance relations.

Definition 3.3.9 Let FSM_s be an FSM of a specification and FSM_i be an FSM of an implementation (both FSMs are connected and minimal; here we mean that they are LTSs derived from FSMs). Let T_{UIO} and T_{UIOv} be the tests suites produced by the procedure **TestCase** using an UIO and an UIOv method, respectively for the step 3) of the procedure. Then

1. FSM_i **UIO** $FSM_s \Leftrightarrow FSM_i$ **passes** T_{UIO};

2. FSM_i **UIOv** $FSM_s \Leftrightarrow FSM_i$ **passes** T_{UIOv}.

The main ideas for expressing the **UIO** relation in *ioco* terms are the following: 1) the implementation conforms to the specification if it passes T_{UIO}; 2) the implementation passes T_{UIO} if every trace from T_{UIO} is contained in the set of traces of the implementation and this is true if 3) for every trace in T_{UIO} the implementation produces after every input prefix of that trace (a prefix which ends into an input) the same output as the specification does ($\forall i_1 o_1 ... i_n o_n \in T_{UIO}, j \leq n$: $out(FSM_i$ **after** $i_1 o_1 ... i_j) = out(FSM_s$ **after** $i_1 o_1 ... i_j)$). Please note that the specification and the implementation are FSMs, so they can produce only one output after an input; for this reason we put $=$ in place of $\subseteq$. The same holds for the **UIOv** relation.

It is quite easy to prove that an UIO algorithm generates sound tests and that it is exhaustive with respect to its correspondent **UIO** conformance relation. The exhaustiveness is a result of the definition itself and the soundness is easily proved because any test case is part of the set of all test cases. The same holds for the UIOv algorithm. Because of the completeness, an UIOv algorithm has the same error detection power as its correspondent **UIOv** relation. This observation is needed in the discussion below.

The authors of UIOv ([VCI90]) show that their method can verify whether the FSM of an IUT is the same as the FSM of a specification, i.e. UIOv can detect any missing and erroneous state and I/O of an IUT. This happens under the assumptions on which the UIOv method is applied: A) the specification FSM is connected, B) the specification FSM is minimal and C) the number of the states of the implementation is less than or equal to the number of the states of the specification. Then it can be said that an UIOv algorithm, and consequently its **UIOv** relation, can verify whether FSM_i is the same or not as FSM_s. By the same we mean that there is a bijection between the two FSMs: $\exists f : S_s \rightarrow S_i$ such that 1) f is a bijection and 2) if $s \xrightarrow{l} t$ is a transition of FSM_s, then $f(s) \xrightarrow{l} f(t)$ is a transition of FSM_i, where S_s and S_i are the state sets of the specification and the implementation, respectively. This bijection is used in the proof of Theorem 3.3.15.

In the steps enumerated above we used for a trace the notion of the set of its prefixes which end into inputs. Now let us define the **prefix** binary relation on traces.

Definition 3.3.10 Let L be a label set. Then **prefix** is defined to be the smallest relation satisfying:

1. let $t \in L^*$, then ϵ and t are **prefix** of t;
2. let $t, t' \in L^*$ and $a \in L$, then
   t **prefix** $t' \Rightarrow t$ **prefix** $t'a$.

Because we usually use the **prefix** on sets of traces, we will define the prefix closure operator PC.

Definition 3.3.11 Let $S \subseteq L^*$ be a set of traces. Then, the prefix closure operator $PC : \mathcal{P}(L^*) \rightarrow \mathcal{P}(L^*)$ is: $PC(S) = \{t \in L^* \mid \exists t' \in S : t \textbf{ prefix } t'\}$

The last element of a trace is defined below.

Definition 3.3.12 Let L be a label set. Then, the function $last : L^* \setminus \{\epsilon\} \rightarrow L$ is:

1. let $a \in L$ then $last(a) = a$;
2. let $t \in L^*$ and $a \in L$, then $last(ta) = a$.

The set of prefixes which end in inputs (input-ended prefixes by short) of a set of traces is defined as in the following.

Definition 3.3.13 Let T be a set of traces. Then the set of its prefixes P_T which end in inputs is:
$P_T = \{t \in PC(T) \mid last(t) \in I\}$.

Now we have all the ingredients defined for expressing the **UIO** and **UIOv** relations in *ioco* terms. This will be done in the following subsection.

3.3.3 Classification of the UIO, UIOv and iocofsm conformance relations

Lemma 3.3.14 gives a translation of the **UIO** and **UIOv** conformance relations in *ioco* terms. Theorem 3.3.15 and Theorem 3.3.16 use the results of the lemma to relate the **iocofsm**, **UIO** and **UIOv** conformance relations.

In Lemma 3.3.14 we build two sets of traces F_{UIO} and F_{UIOv} such that: (FSM_i **UIO** FSM_s iff FSM_i $\textbf{iocofsm}_{F_{UIO}}$ FSM_s) and (FSM_i **UIOv** FSM_s iff FSM_i $\textbf{iocofsm}_{F_{UIOv}}$ FSM_s). Unsurprisingly, the set of traces F_{UIO} is the set of the input-ended prefixes of the set T_{UIO} ($F_{UIO} = P_{T_{UIO}}$); the same holds for

F_{UIOv}.

Example For the specification from Figure 3.1 the test suite T_{UIO} is represented in Figure 3.2 (b) and is the set of traces $\{a/b, a/b.a/c.a/b, a/b.d/e.d/e.a/b, d/e.d/e.a/b, a/b.d/e.a/c.a/c.a/b\}$. In this set we did not put the $r/null$ sequence because it is implicitly understood that before running each trace, FSM_i and FSM_s should be in the initial state so that a new test case can be run. Now the set of its input-ended prefixes is $F_{UIO} = \{a\} \cup \{a/b.a/c.a, a/b.a, a\} \cup \{a/b.d/e.d/e.a, a/b.d/e.d, a/b.d, a\} \cup \{d/e.d/e.a, d/e.d, d\} \cup \{a/b.d/e.a/c.a/c.a, a/b.d/e.a/c.a, a/b.d/e.a, a/b.d, a\} = \{a, a/b.a/c.a, a/b.a, a/b.d/e.d/e.a, a/b.d/e.d, a/b.d, d/e.d/e.a, d/e.d, d, a/b.d/e.a/c.a/c.a, a/b.d/e.a/c.a, a/b.d/e.a\}$. For the **UIOv** relation $F_{UIOv} = F_{UIO} \cup \{d/e.a\}$.

Lemma 3.3.14 *Let FSM_i be an implementation and let FSM_s be a specification (so here we mean that they are LTSs derived from FSMs). Then there exist two sets of traces F_{UIO} and $F_{UIOv} \in L^*$ such that:*

1. FSM_i **UIO** $FSM_s \Leftrightarrow FSM_i$ $\textbf{iocofsm}_{F_{UIO}}$ FSM_s;
2. FSM_i **UIOv** $FSM_s \Leftrightarrow FSM_i$ $\textbf{iocofsm}_{F_{UIOv}}$ FSM_s.

Proof: According to Definition 3.3.9 and Definition 3.3.6:

1. FSM_i **UIO** $FSM_s \Leftrightarrow FSM_i$ **passes** $T_{UIO} \Leftrightarrow T_{UIO} \subseteq traces(FSM_i)$;
2. FSM_i **UIOv** $FSM_s \Leftrightarrow FSM_i$ **passes** $T_{UIOv} \Leftrightarrow T_{UIOv} \subseteq traces(FSM_i)$.

Below we will continue the proof for point 1) of the lemma. Let $\sigma \in T_{UIO}$ be a trace. Let P_σ be the set of its input-ended prefixes. The trace σ is contained in $traces(FSM_s)$ and FSM_s is a deterministic FSM. Then for every $\sigma' \in P_\sigma$, the set $out(FSM_s\ \textbf{after}\ \sigma')$ contains only the output which is in σ after the subtrace σ'. This output is denoted as $out(\sigma')$. We have that $out(FSM_s\ \textbf{after}\ \sigma') = out(\sigma')$. Then:

$$\sigma \in traces(FSM_i) \text{ iff } \forall \sigma' \in P_\sigma : out(FSM_i\ \textbf{after}\ \sigma') = out(\sigma') = out(FSM_s\ \textbf{after}\ \sigma')$$

In other words, a trace of the specification is also a trace of the implementation if each output produced by the implementation after performing an input prefix of that trace is specified.

Let us consider $F_{UIO} = \cup_{\sigma \in T_{UIO}} P_\sigma = P_{T_{UIO}}$. Then:

$$T_{UIO} \subseteq traces(FSM_i) \text{ iff } \forall \sigma' \in F_{UIO} : out(FSM_i\ \textbf{after}\ \sigma') = out(\sigma') = out(FSM_s\ \textbf{after}\ \sigma')$$

And this means that:

$$FSM_i\ \textbf{UIO}\ FSM_s \Leftrightarrow FSM_i\ \textbf{iocofsm}_{F_{UIO}}\ FSM_s;$$

In a similar way, the point 2) of the lemma is proved if we take $F_{UIOv} = P_{T_{UIOv}}$. □

We will compare conformance relations such as **UIO**, **UIOv** and **iocofsm**. We want to write for example **UIOv** $\geq$ **UIO** with the intuition that **UIOv** is more powerful than **UIO**.

We want to make such comparison statements under certain conditions, for example considering only the situation where the specification is connected, minimal and where the size of the state space of the implementation does not exceed that of the specification. We write

UIOv $\geq$ **UIO**

to mean that whenever FSM_i **UIOv** FSM_s we find that FSM_i **UIO** FSM_s. The same meaning has the notation: **iocofsm** $\geq$ **UIO**. By **UIOv** $\sim$ **iocofsm** we mean: **iocofsm** $\geq$ **UIOv** $\wedge$ **UIOv** $\geq$ **iocofsm**.

Theorem 3.3.15 *Let FSM_i be an implementation and let FSM_s be a specification. Let us assume that*

a) FSM_s *is connected;*

b) FSM_s *is minimal;*

c) the number of the states of FSM_i *is less than or equal to the number of the states of* FSM_s*.*

Then:

1. **UIOv** $\geq$ **UIO***;*

2. **iocofsm** $\geq$ **UIO***;*

3. **UIOv** $\sim$ **iocofsm**.

Proof: **UIO** $=$ **iocofsm**$_{F_{UIO}}$ and **UIOv** $=$ **iocofsm**$_{F_{UIOv}}$. For the **UIO** relation, **iocofsm**$_{F_{UIO}}$ has less discriminating power than the **iocofsm** $=$ **iocofsm**$_{traces(FSM_s)}$ relation (because $F_{UIO} \subseteq traces(FSM_s$)) and the **UIOv** relation (because $F_{UIO} \subseteq F_{UIOv}$). Then **UIOv** $\geq$ **UIO** $\wedge$ **iocofsm** $\geq$ **UIO** and so the first two points of the theorem are proved.

For proving point 3), we should remember that **UIOv** $\sim$ **iocofsm** means **iocofsm** $\geq$ **UIOv** $\wedge$ **UIOv** $\geq$ **iocofsm**. The first part **iocofsm** $\geq$ **UIOv** is true because **UIOv** has less discriminating power than **iocofsm** (because $F_{UIOv} \subseteq traces(FSM_s)$).

For the second part of the conjunction, let us assume that it is not true. Then $\exists FSM_i, FSM_s$ such that

- FSM_i **UIOv** FSM_s
- $\neg(FSM_i$ **iocofsm** $FSM_s)$

Then let σ be a trace such that $out(FSM_i$ **after** $\sigma) \neq out(FSM_s$ **after** $\sigma)$. If FSM_i **UIOv** FSM_s, then there is a bijection between FSM_i and FSM_s (see Section 3.3.2): $\exists f : S_s \rightarrow S_i$ such that 1) f is a bijection and 2) if $s \xrightarrow{l} t$ is a transition of FSM_s, then $f(s) \xrightarrow{l} f(t)$ is also a transition of FSM_i, where S_s and S_i are the state sets of the specification and the implementation, respectively. Let $\sigma = i_1/o_1...i_n$ with $n \in \mathbf{N}$. Then, this trace is executed in FSM_i in the following way: $f(s_0) \xrightarrow{i_1} ... \xrightarrow{i_n} f(s_{2n-1})$, where $s_0, ..., s_{2n-1}$ are the states of FSM_s through which σ is going. According to the bijection, if $s_{2n-1} \xrightarrow{o_n} s_{2n}$ is a transition of FSM_s, then $f(s_{2n-1}) \xrightarrow{o_n} f(s_{2n})$. Now we have contradiction with $out(FSM_i$ **after** $\sigma) \neq out(FSM_s$ **after** $\sigma)$, because the output is the same. Because of the contradiction, the second part of the conjunction is true and, then, it can be concluded that **UIOv** $\sim$ **iocofsm**. □

An example in which $\neg(FSM_i$ **iocofsm** $FSM_s)$ and $\neg(FSM_i$ **UIOv** $FSM_s)$, but FSM_i **UIO** FSM_s is given in Figure 3.1. This example shows that in general the **UIOv** and **iocofsm** relations are more powerful than the **UIO** relation in detecting errors in IUTs.

In this point, we need to make an observation. The proof of the point 3 of the theorem from above relies on the proofs from [VCI90] which are not that precise. Because our goal is not to improve these proofs, but to compare algorithms and tools, we will rely on [VCI90] as we did for TGV when we rely on the proofs from [JM99, FJJT96]. Moreover, to the best of our knowledge there are no counter-examples to the claims from [VCI90]. (But, as a remark, mistakes can be made: the early versions of TGV work with **ioco** and not **ioconf**, although the authors call it **ioconf** [FJJT96]. Later they admitted they were wrong. Then they called it **ioco** [JM99].)

The following theorem relates the **UIOv** and **iocofsm** conformance relations in the general case in which the assumption *c)* does not hold. At the end of this subsection a summary of the classification is given. In the theorem by **iocofsm** $>$ **UIOv** we mean that:

1. $\forall FSM_i, \forall FSM_s : FSM_i$ **iocofsm** $FSM_s \Rightarrow FSM_i$ **UIOv** FSM_s,

2. $\exists FSM_i, \exists FSM_s : \neg(FSM_i$ **iocofsm** $FSM_s) \wedge FSM_i$ **UIOv** FSM_s.

Theorem 3.3.16 *If* $\neg c)$ *then* **iocofsm** $>$ **UIOv**.

Proof : For proving that **iocofsm** $>$ **UIOv** when *c)* does not hold, we should show that both 1) and 2) are true. The first part 1) is true because **UIOv** has less discriminating power than **iocofsm** (because **UIOv** $=$ **iocofsm**$_{F_{UIOv}}$ and $F_{UIOv} \subseteq traces(FSM_s)$).

The second part 2) is proved by a counter example. Let us consider the implementation i from Figure 3.4 and the specification *Spec* from Figure 3.1. Then: i **UIOv** *Spec* (i passes all the UIOv test cases from Figure 3.2), but $\neg(i$ **iocofsm** *Spec*) (i can be detected with the trace $a/b.a/c.a/b.a/c \in traces(Spec)$). □

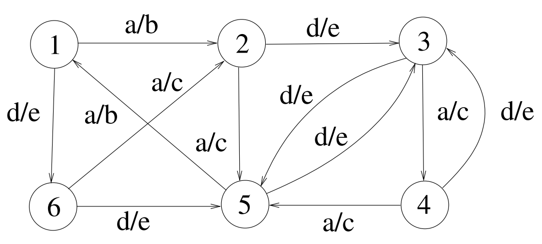


Figure 3.4: A faulty IUT.

The summary of the classification is given in the following table

conditions	1.	2.	3.
a) $\wedge$ *b)* $\wedge$ *c)*	**UIOv** $>$ **UIO**	**iocofsm** $>$ **UIO**	**iocofsm** $\sim$ **UIOv**
arbitrary ($\neg$*c)*)			**iocofsm** $>$ **UIOv**

In concluding the section, it is important to note that, due the heavy constraints (determinism, limit for the implementation's number of states) the UIO and UIOv algorithms are well-known not to find all errors. Taking into account that **iocofsm** is only a part of the original **ioco**, part which is defined for LTSs constructed from deterministic, connected and minimal FSMs and that **iocofsm** is more powerful than **UIO** and **UIOv**, we can conclude that the *ioco* algorithms are in general more powerful than the UIO and UIOv algorithms.

When the constraints hold the UIOv algorithms have the same detection power as the **iocofsm** conformance relation, but UIO is less powerful than the **iocofsm** conformance relation and UIOv algorithms. One conclusion might be that, under the constraints, *ioco* algorithms have the same detection power as the UIOv algorithms. For this to hold in a practical sense, the following assumption is needed. The assumption is that TorX and TGV can be modified such that they can work on LTSs derived from FSMs and that these modifications are based on the **iocofsm** conformance relation. Such a modification is not difficult to be imagined for TorX, but it does not exist in practice, as a tool. The same holds for TGV. Under these considerations we can observe that this equality, in theory, might happen but it is based on heavy constraints and on a theoretical assumption which is not implemented in practice.

3.4 Autolink

Autolink is the name of the algorithm which is integrated in the TAU/Telelogic tool-set ([SEG$^+$98]) for generating test cases. Autolink supports the automatic generation of TTCN test suites based on an SDL specification and a test purpose described in MSC. It is widely used because it is integrated in a commercial tool (TAU). It can be used in a batch-oriented way.

It is interesting to remark that a central topic which is treated by this thesis, namely test selection, is also addressed by the creators of Autolink. Autolink uses test purposes for doing test selection. By defining test purposes and then generating a test suite, a set of representative behaviours to be tested is selected. Test purposes combine the guidance provided by the human intelligence with the power of automation. This is an advantage of the Autolink approach. Our approach is different from the one of Autolink in the following way. We tried to fully automate the test selection by the use of selection heuristics and trace distances (see Chapter 8). Our approach is a step ahead in the integration of the test generation and the execution into one on-the-fly process. A disadvantage can be the fact that the combination of human intelligence and automation can be lost in a fully automated test selection, and this combination is important sometimes in selection (for example when it is known apriori for an IUT which specific set of behaviours should be tested).

Since Autolink does not have an explicit conformance relation on which it is based, we find it necessary to try to reconstruct a conformance relation for it and in this way to consolidate the conformance foundation of it. Based on this conformance relation we will compare Autolink to the other algorithms: TorX, TGV and UIO (UIOv) algorithms. We will express the conformance relation in terms of *ioco* theory.

This section is organized in the following way: Subsection 3.4.1 describes the Autolink algorithm; Subsection 3.4.2 gives the translation of the Autolink algorithm in *ioco* terms; Subsection 3.4.3 gives the classification of Autolink in comparison to other algorithms.

3.4.1 The Autolink algorithm

Autolink is an algorithm for test derivation which has as inputs: 1) the specification which is written in SDL and 2) the test purpose which is expressed in MSC. It produces as an output a test case represented in TTCN.

Before explaining the algorithm we should say some words about the MSC [IT93] which represents the test purpose. One thing to remember is that semantically an MSC describes a partial ordering of the events. This means that the events of an instance are partially ordered; between two instances there is not a temporal order, except for the ordering induced by the messages. Autolink looks at the system under test as a black box to which signals are sent and received via different PCOs (Points of Control and Observation). The system under test which is described by the SDL specification is one instance of the test purpose MSC. The PCOs are the other instances of the MSC. For generating tests, Autolink looks at the events which happen at the PCO instances. So it does not look at the partial order of the events of the instance of the specification; it looks only at the partial order of the events of the PCO instances. So from an MSC, a set of traces can be formed (to be checked that these are traces of the specification) by taking all the possible linearizations of the events which appear in the MSC and which respect the partial order of the PCO instances and without considering the partial order of the specification instance.

Example Let us consider the MSC from Figure 3.5. In this MSC the specification is called *Example* and it forms one instance of the MSC. The other instances are the PCOs: A and B. An obvious trace to be checked is $acbd$. This trace respects the partial order of the specification instance. But at first, Autolink does not care about this partial order. It will also use traces like: $abcd$, $abdc$ which do not respect the partial order of the *Example* instance (in the generated test, Autolink will keep only the traces which are a trace of the specification). So from this MSC a set of traces can be generated and it is $\{acbd, abcd, abdc, bdac, badc, bacd\}$.

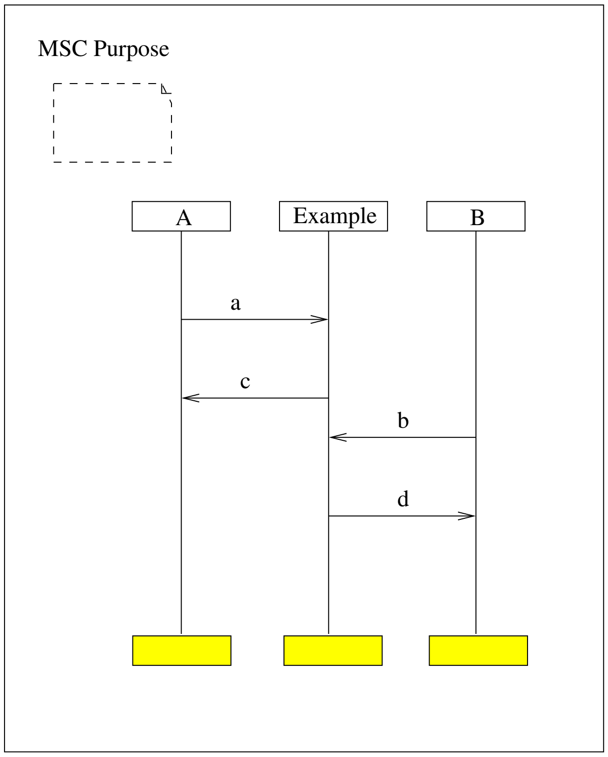


Figure 3.5: A test purpose MSC.

For generating a TTCN test, Autolink considers only a subset of traces from the set of traces generated as above. This is because Autolink applies some further restrictions. For example, Autolink uses priorities between send events and receive events. The authors of Autolink ([SEG+98]) suppose that the environment always sends signals to the system as soon as possible, whereas receive events occur with an undefined delay. This is motivated by the assumption that the tester, which is represented by the environment, is faster than the system under test. This assumption can be true for a batch-oriented test generation, because the test execution is quite fast. For an on-the-fly test generation, this assumption does not usually hold because test generation, which is combined with test execution, is time consuming. For Autolink which works batch-oriented, this assumption is considered to hold and therefore the send event has a higher priority than the receive event in the default configuration setting of Autolink. All the traces from the set of traces of the MSC which do not respect this rule are removed from it. This will work as follows in our example: the set of inputs is $\{a, b\}$ and the set of outputs is $\{c, d\}$; the traces *acbd* and *bdac* are removed because one of the outputs c and d happens before one of the inputs b and a and this contradicts the rule that the input event has a higher priority than the output event; so the subset obtained is $\{abcd, abdc, badc, bacd\}$. Some other restrictions exist. They are parameterized in the options configuration of Autolink.

In our explanation of Autolink we will not enter into technical details such as how to generate an MSC test purpose or how to configure the options of the Autolink algorithm. Also we will only consider MSC without references because this is general enough to cover a number of important aspects of Autolink.

The steps performed by Autolink can be described shortly in the following way:

1. *Validation:* the MSC which represents the test purpose is validated against the SDL specification (this means that at least one of the traces from the set of traces of the MSC must be a valid trace of the SDL specification);

2. *Generation:* the traces formed from the validated MSC which respect the options configuration of Autolink are checked against the SDL specification with a *state space exploration* algorithm and the TTCN test case is formed by all the MSC traces which are also traces of the SDL specification.

In the TTCN test case generated by Autolink a Timeout event gets an **inconclusive** verdict. When an unexpected output is produced by the IUT (OtherwiseFail event) a **fail** verdict is given by the TTCN test case. The traces of the specification which are not contained in the set of traces of the MSC (we will call them uninteresting traces) are cut and thus **inconclusive** verdicts are assigned to them. For explaining how the cutting process works in Autolink, we should first introduce the notion of *global states* of an SDL system because Autolink works with transitions between global states. For a good understanding of this notion, first we will introduce an example of an SDL specification. Using this SDL specification, we will also exemplify the generation of a TTCN test case with Autolink.

Example From the SDL specification *Example*, only the process which describes the behaviour of the *Example* specification is shown in Figure 3.6. The process can receive from the environment the input a via channel A or input b via channel B. After receiving input a, it can send to the environment output c (via channel A) and it returns in the initial state *State 1*. A similar thing happens if input b is received: output d is sent to the environment (via channel B) and it returns to *State1*.

For understanding Autolink, one important aspect to be considered is the channel buffer mechanism. An SDL specification has queues associated with channels. When more signals are present in a queue, they will be consumed in the same order in which they arrived, i.e. in FIFO order. Executing a transition between two states means performing an input which is allowed in a state and sending all the produced outputs to queues of channels. These outputs can be stored and they are not necessarily consumed immediately after their production. After reaching a new state, the SDL specification can perform another input and after that send the correspondent outputs to queues of channels. In this way a queue can have more than one signal stored at a given time. Some of the outputs produced are outputs send to environment or internal signals. An internal signal is seen as an output for the process which sends it and as an input for the process which receives it. Therefore there are three kinds of channel queues: inputs queues, output queues and internal channel queues. From the point of view of the process which sends an internal signal, an internal channel queue is seen as an output queue. From

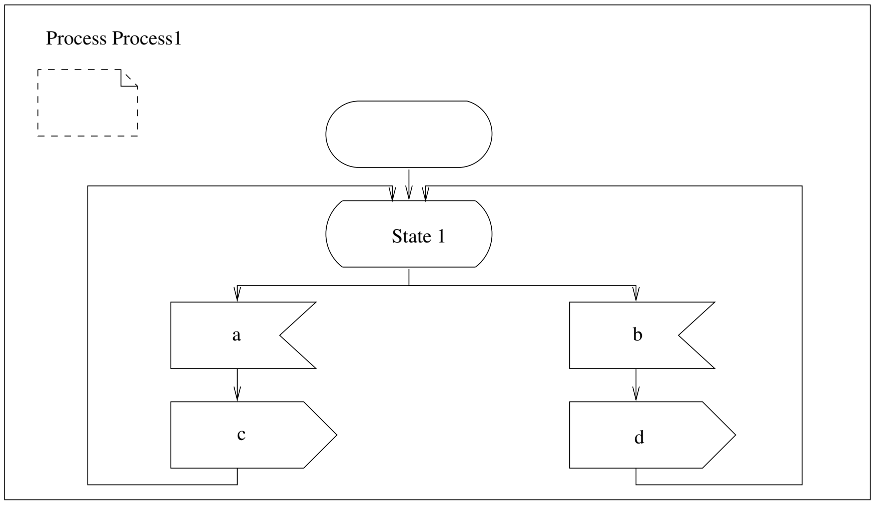


Figure 3.6: The SDL specification *Example*.

the point of view of the process which receives an internal signal, an internal channel queue is seen as an input queue. With these things being said, the mechanism works as follows: an SDL specification, which can be seen as a collection of processes, has a set of states in which it can be at a given time, each of these states corresponding to a process which is active at that time. Relevant information which characterizes the global state in which the SDL system is at a given time is the set of states of active processes (active states by short), the outputs which are contained in channel queues, the inputs which are received and not consumed yet in the input queues and the internal signals from the internal channel queues. Now, the SDL specification can perform any input received from the environment which is allowed in an active state (all these inputs form the valid input set of the global state) or send any output (the output is said to be *consumed*) which becomes present at the top of a channel queue or perform any internal signal which is allowed in an active state and present at the top of a channel queue. The execution of an output sent to environment does not change the set of active states in which the SDL specification can be (it remains the same). It only removes one output from a channel queue. But this removal changes the content of that queue and therefore the global state of the SDL system is different from the previous one. If an internal signal is consumed by one of the active processes, its consumption can be seen as an input execution because its reception will cause the SDL process to perform a transition between two states. Now we will explain the input execution. When executing an input (received from the environment or as an internal signal from another process), the state from which it was performed is removed from the set of active states and the reached state is added. The input will be deleted from its queue and the outputs which are produced are added to channel queues. If an input is received and it is not contained in the valid input set, it will be stored in an input queue. These changes characterize the new global state of the SDL system. When generating tests, Autolink ([KGHS98]) works with global states. An allowed behaviour of the SDL system is formed by allowed transitions between global states.

Now let us see how this works for our example (see also Figure 3.9). In this example, the SDL system is formed by one process. Only this process is active at any given time and for this reason the set of active states is {*State1*}. This SDL specification has two buffered channels: A and B. Via A, the SDL system can receive inputs or send outputs. For this reason there are two queues for A: one for receiving inputs from the environment, denoted as A_I, and one for sending outputs, denoted as

A_U. The same holds for B: the queue B_I is for inputs and the queue B_U is for outputs. As explained above, the relevant information which characterizes a global state of an SDL system is the set of active states and the outputs which are present in channel queues. There are no internal channels. Because this SDL system is input complete in *State1* it results that all the input queues are empty in all global states. Therefore we do not refer below to input queues. For the system *Example*, we will show how this relevant information is changed from one global state to another. In the initial global state of *Example* the set of active states is {*State1*} and the output queues of A and B are empty ($A_U = \langle\rangle$ and $B_U = \langle\rangle$). In this state, the SDL system can perform one of the inputs a or b sent through one of the channel queues A_I or respectively B_I. Performing input a will send the output c to the queue for outputs of channel A, i.e. to A_U. This output is stored and it is not necessarily visible immediately after a. In the new global state of *Example*, the set of active states is unchanged (it is {*State1*}) and the queues for outputs are $A_U = \langle c\rangle$ and $B_U = \langle\rangle$. In this new global state, it is possible to perform an input, a or b, or the output c. Performing, for example, b will produce the following changes: B_U becomes $\langle d\rangle$ and {*State1*} and A_U remain unchanged. Two next transitions can be cd or dc. In this way $abcd$ or $abdc$ are valid traces of the specification. The execution of other transitions proceeds in a similar way as described above.

After we described the channel buffer mechanism we will explain below how the cutting process works in Autolink (see also Figure 3.7). There are two cases for generating an **inconclusive**. In the first case, in the test generation process of Autolink a trace from the set of traces of the MSC, reaches different global states of the specification. In a global state reached, there can be outputs which are not contained in any subsequent traces of the MSC, or, in other words outputs which do not satisfy the MSC test purpose. By subsequent traces we mean traces which appended to the subtrace which led to these outputs form valid traces of the MSC. These outputs which do not satisfy the MSC test purpose are specified and can be produced by an IUT. They get **inconclusive** verdicts ([SKGH97]). Another source of **inconclusive** verdicts is an incomplete **pass** trace. Some traces of the MSC are only partial valid traces of the specification. This means that only a prefix fits to the specification. The prefix is called an incomplete **pass** trace. The rest of the trace is not considered. The prefix (incomplete **pass** trace) gets an **inconclusive** verdict ([SKGH97]).

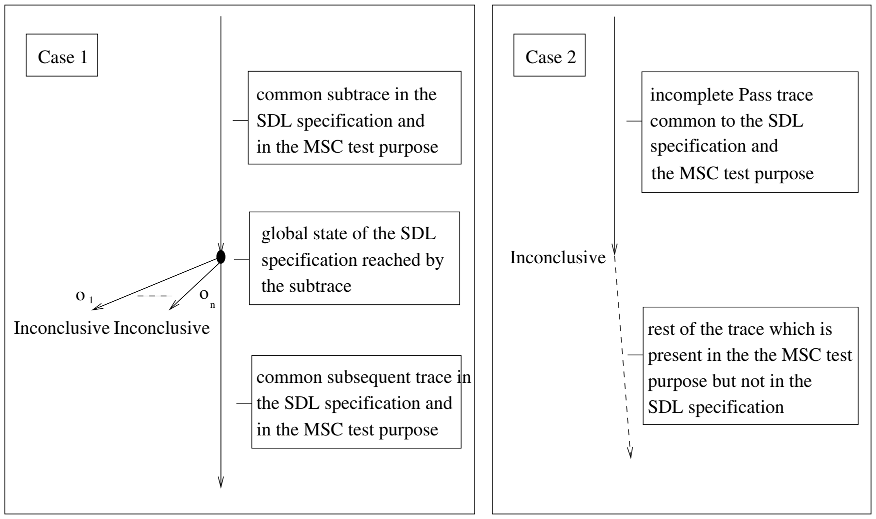
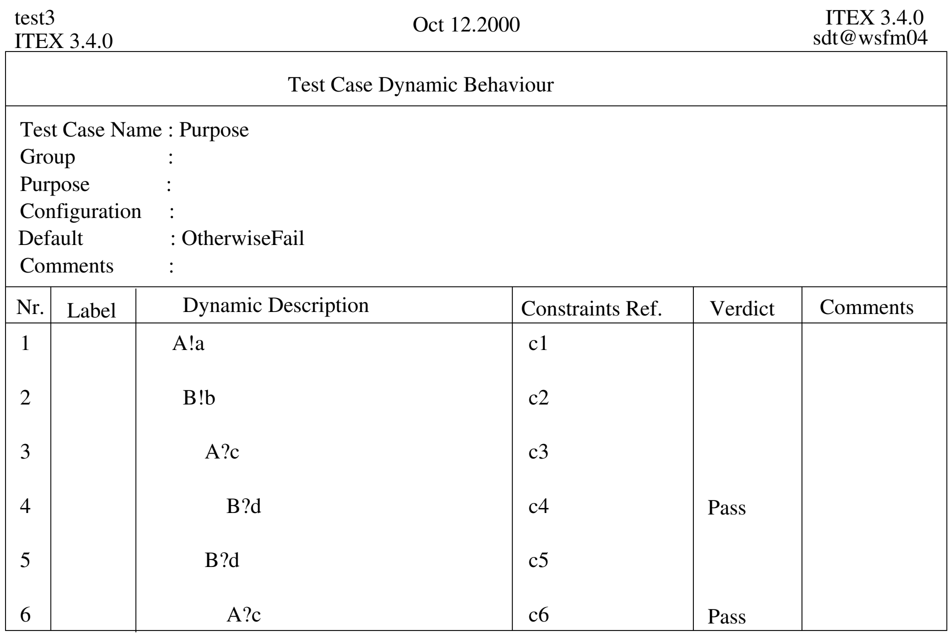

Case 1
common subtrace in the SDL specification and in the MSC test purpose
global state of the SDL specification reached by the subtrace
o1
on
Inconclusive Inconclusive
common subsequent trace in the SDL specification and in the MSC test purpose
Case 2
incomplete Pass trace common to the SDL specification and the MSC test purpose
Inconclusive
rest of the trace which is present in the the MSC test purpose but not in the SDL specification

Figure 3.7: The two cases for generating an **inconclusive**.

We will illustrate how a TTCN test case can be generated with Autolink for the SDL specification

Example. The test purpose is described in the MSC of Figure 3.5. This MSC is a valid test purpose because there is at least one trace of the *Example* specification which is also part of the MSC set of traces (for example the trace $abcd$).

Using the default options configuration, Autolink generated the TTCN test case of which the dynamic behaviour is presented in Figure 3.8. In this TTCN test, there are two traces which are present in the set of traces of MSC (reduced according to the restrictions imposed by the default options configuration) and which are also part of the set of traces of the *Example* specification; these are $abcd$ and $abdc$. Autolink merges these traces into a TTCN tree. After performing one of these traces, an IUT will get a **pass** verdict.



test3
ITEX 3.4.0

Oct 12.2000

ITEX 3.4.0
sdt@wsfm04

Test Case Dynamic Behaviour

Test Case Name : Purpose
Group :
Purpose :
Configuration :
Default : OtherwiseFail
Comments :

Nr.	Label	Dynamic Description	Constraints Ref.	Verdict	Comments
1		A!a	c1		
2		B!b	c2		
3		A?c	c3		
4		B?d	c4	Pass	
5		B?d	c5		
6		A?c	c6	Pass	

Dynamic Part

Figure 3.8: The test generated with Autolink.

3.4.2 The translation of Autolink in *ioco* terms

The first concern regarding the translation of Autolink in terms of *ioco* theory is how to transform the SDL system into a labelled transition system. This can be done by simply assuming that the labelled transition system of an SDL system is the unfolding tree of the SDL system. We devote one paragraph to explain this notion of unfolding tree.

The transitions in the tree are labelled with the name of the signals (inputs, outputs or internal signals) and its states contain the triple ⟨*set of active states, set of channel queues, unique identifier*⟩. The *Root* state of the tree which is also the initial state of the labelled transition system is ⟨*set of initial active states, set of channel queues (all empty), unique identifier* (0)⟩. The unfolding tree is constructed in the following way. Let $S = \langle St, SC, Id \rangle$ be a state in the tree (or a global state), where *St* is the set of active states, *SC* is the set of channel queues and *Id* the unique identifier of S. We assume that the state labels of the various processes to be disjoint. Let I be a valid input or an internal signal which can be executed in S. Let $s \in St$ be an active state where I can be performed. Let $s \overset{I O_1 ... O_n}{\longrightarrow} s'$ be the correspondent transition of the correspondent process of the SDL system, where $n \in \mathbf{N}$, $O_1 ... O_n$ are either outputs (sent to environment) or internal signals. Then the new global state S' is given by $S \xrightarrow{I} S' = \langle St \setminus \{s\} \cup \{s'\}, Add(SC', O_1, ..., O_n), Id' \rangle$, where *Id'* is a new fresh identifier and *Add(SC'*, $O_1, ..., O_n$) is the result of adding all the signals $O_1, ..., O_n$ to the queues from *SC'* in that order. If no output was produced, then, in place of *Add(SC'*, $O_1, ..., O_n$) we will have *SC'*. The set of channels queues *SC'* equals *SC* if I was not taken from a queue from SC (the sending of I from the environment and its consumption is seen as being one transition). Otherwise *SC'* is *Rem(SC*, I) where *Rem* is a procedure which removes input I from its correspondent queue in *SC*. Let I be a non-valid input sent by environment. Then the new global state is given by $S \xrightarrow{I} S' = \langle St, Add(SC, I), Id' \rangle$, where *Id'* is a new fresh identifier and *Add(SC*, I) adds the input I in a queue in *SC*. Let O be an output that is at the top of a queue from *SC*. Then the new global state is given by $S \xrightarrow{O} S' = \langle St, Rem(SC, O), Id' \rangle$, where *Id'* is a new fresh identifier and *Rem(SC*, O) removes the output O from its correspondent queue in *SC*.

A complete formal definition of the semantics of SDL is beyond the scope of this thesis. But it is clear that other language constructs of SDL could be handled along the same lines (e.g. saving signals, timers [CCI92]).

Definition 3.4.1 The labelled transition system $M' = \langle S, L, \rightarrow, s_0 \rangle$ of an SDL specification is the unfolding tree of the SDL specification where:

1. S is formed by the set of states of the unfolding tree;
2. the set of labels L is formed by the union of the set of inputs with the set of outputs of the SDL system and with $\{\tau\}$;
3. the transitions of $\rightarrow$ are the transitions of the unfolding tree; internal signals are transformed in τ;
4. the initial state s_0 is the *Root* state of the unfolding tree.

Please note that an LTS derived from an SDL system is input complete, i.e. any input sent from environment (valid or non-valid) can be received in any state. In practise, because there are no queues with unbounded length, it is usual to add extra transitions (using the construct * – any) for consuming the inputs which are not valid. In this way any input can be immediately consumed in any global state of the specification. For the sake of simplicity, without loss of generality, we will restrict ourself to this kind of SDL systems. For these systems the input queues are empty in all the global states.

Example For the SDL system of Figure 3.6, its labelled transition system is represented in Figure 3.9. The initial state, denoted as $\langle \{State1\}, \{A_U = \langle\rangle, B_U = \langle\rangle\}, 1\rangle$ in the figure, has the set of active states $\{State1\}$ and the output queues, A_U and B_U, empty. Because the input queues A_I and A_U are always empty in any global state, for the sake of simplicity, we did not include them in the information related to the global states. From this state, it is able, after performing the input a, to arrive in the state $\langle \{State1\}, \{A_U = \langle c\rangle, B_U = \langle\rangle\}, 2\rangle$ and, after performing the input b, to reach the state $\langle \{State1\}, \{A_U = \langle\rangle, B_U = \langle d\rangle\}, 3\rangle$. In the state $\langle \{State1\}, \{A_U = \langle c\rangle, B_U = \langle\rangle\}, 2\rangle$, the SDL system can perform the inputs b or a or send the output c. In the state $\langle \{State1\}, \{A_U = \langle\rangle, B_U = \langle d\rangle\}, 3\rangle$ the SDL system can perform the inputs b or a or send the output d. The construction of the labelled transition system of this SDL system proceeds in a similar style as explained above.

The elements from the name of the states signify, taking for example the state $\langle\{State1\}, \{A_U = \langle\rangle, B_U = \langle d\rangle, 1\rangle\}, 3\rangle$: 1) the set of the active states, $\{State1\}$; 2) the set of channel queues which has four elements: two empty input queue (not represented) and the output queue of channel A which is empty ($A_U = \langle\rangle$) and the output queue of channel B which contains the output d ($B_U = \langle d\rangle$); 3) the number 3 is the identifier which uniquely identifies this state.

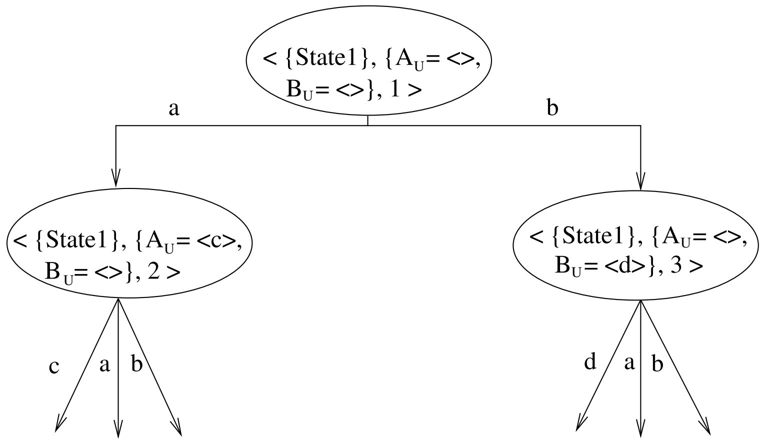

< {State1}, {A_U= <>, B_U= <>}, 1 >
a
b
< {State1}, {A_U= <c>, B_U= <>}, 2 >
< {State1}, {A_U= <>, B_U= <d>}, 3 >
c
a
b
d
a
b

Figure 3.9: The labelled transition system of the SDL specification.

Now we are going to express Autolink in *ioco* terms. As we explained in Section 3.4.1, an MSC test purpose can be replaced by a set of traces. For an MSC M, we will denote its set of traces as $S(M)$. In M we assume that the first instance is the system under test and the rest of the instances are the PCOs; with this assumption, without loss of generality, we do not put the list of PCOs as a parameter of $S(M)$. The options configuration of Autolink which selects only a subset from the set of traces of the test purpose MSC, can be seen as a predicate P on traces. The TTCN test case will be represented as a set of traces. We will not enter in technical details regarding how Autolink implements the cutting of the uninteresting traces which get **inconclusive** verdicts (see Section 3.4.1). We will assume that this finite set of traces is produced by a procedure which is part of Autolink and which we will call *inconcs*. The actual parameters of *inconcs* are: $inconcs(Q, S(M), P)$. The two parameters of the procedure $S(M)$ and P have the meanings as described above. The parameter Q is the labelled transition system of the SDL specification. The set of traces produced by *inconcs* can be expressed as being the union of $\{to' \in L_\delta^* \mid \exists t' \in traces(Q) \cap S(M), o \in L_U : P(t'), to \textbf{ prefix } t', o' \in out(Q \textbf{ after } t), o' \neq o\}$ and $\{t \in traces(Q) \mid \exists t' \in S(M) : P(t'), t' \notin traces(Q), t \textbf{ prefix } t'\}$. The first set of the union is formed by common subtraces of the MSC test purpose and the SDL specification concatenated with specified outputs which are not present in any subsequent trace of the MSC (the first case from Figure 3.7). The second set is formed by incomplete **pass** traces (the second case from Figure 3.7, see also Section 3.4.1). In the formalization above, one should remember that the *out* function is defined on a simple automaton, which is not a suspension one (see Definition 2.1.8) and that the null output δ can be contained in the outputs set produced by *out* (this observation is useful later).

Before expressing *Autolink* in *ioco* terms, because in *ioco* theory there are only **pass** and **fail** verdicts, we will indicate how the **inconclusive** verdicts contained in a test produced by Autolink can be transformed. The **inconclusive** verdict assigned to the uninteresting traces of the specification which are cut by the Autolink algorithm, can be replaced by a **pass** verdict. For the Timeout event, we can see it as a mapping of the null output (δ). Now, if the trace which led to this Timeout event reaches in the labelled transition system of the specification a state which contains only inputs, then the null output (Timeout) is specified. Therefore, its **inconclusive** can be transformed into a **pass**. If all states reached by the trace contains outputs, then the null output (Timeout) is unspecified and its **inconclusive** can be replaced by a **fail**. These transformations are reflected in the formalization of the **Autolink** algorithm from below in the following way. First, we remind that the function *out* works with δ outputs. The δ outputs (Timeouts) which gets a **pass** are present in the traces of *inconcs*. The ones which get a *fail* are not considered by the function *out* and, therefore, they are not present in the traces of *inconcs*. The *inconcs* function models also the Timeout events. Now all the outputs which can get a **pass** verdict are present in the set of traces produced by Autolink. The outputs which get a **fail** are not present. In this way, the presence or the absence of an output decides the verdict which is assigned to that output: **pass** or **fail**, respectively. This will be caught in the definition of **passes** (Definition 3.4.2) for Autolink.

Now the Autolink algorithm can be expressed in *ioco* terms in the following way.

Autolink

Inputs:

- the labelled transition system Q of the SDL specification S;
- the set of traces $S(M)$ of the MSC test purpose M;
- the predicate P on traces defined by the option configuration of Autolink.

Output:

- the set of traces T which represents the test case generated.

Body of Autolink

1. *Validation:* $traces(Q) \cap S(M) \neq \emptyset$;
2. *Generation:* $T' = \{t \in traces(Q) \cap S(M) \mid P(t)\} \cup inconcs(Q, S(M), P)$;

   (the union of the set of traces which are common in the test purpose and the specification and the set of traces from the specification which are cut because they are not present in the test purpose and which get **inconclusive** verdicts);

   $T = PC(T')$ (the test is the set of the prefixes of T').

For the rest of this chapter we will refer to the algorithm which is described above as the Autolink algorithm.

Now we will define what is **passes** for the Autolink algorithm. We will not keep Definition 3.3.6 for **passes** because it is only working for labelled transitions systems derived from finite state machines (an FSM which can only produce an output in a state). For a labelled transition system, in general, we will need more requirements. These requirements are added in Definition 3.4.2. We should remember also that Autolink checks the outputs of the implementation only in positions of the traces where outputs exists. With these considerations the new definition of **passes** is:

Definition 3.4.2 Let T be a test suite and w the labelled transition system of an implementation. Let $P_T = \{t \in L^* \mid \exists o \in L_U : to \in PC(T)\}$ be the set of the prefixes from the traces of T which are followed by an output. Let $P_{T,w} = \{to \mid t \in P_T, o \in out(w \textbf{ after } t)\}$ be the set of traces built from every trace from P_T concatenated with every output produced by the implementation w after performing the trace. Then: w **passes** $T \Leftrightarrow P_{T,w} \subseteq T$.

Example Let us consider the SDL specification *Example* from Figure 3.6 and the TTCN test suite from Figure 3.8. The set of traces T of this test suite is $\{a, ab, abc, abd, abcd, abdc\}$ and the set P_T is $\{ab, abc, abd\}$. Now let us assume the implementation *Imp* from Figure 3.10. *Imp* produces after performing the trace ab the output set $\{c, d'\}$, after abc the set $\{d'\}$ and abd is not even a valid trace of this implementation. The set $P_{T,Imp}$ is $\{abc, abd', abcd'\}$. Now $\{abc, abd', abcd'\} \not\subseteq \{a, ab, abc, abd, abcd, abdc\}$ which means that $P_{T,Imp} \not\subseteq T$. As can be seen also in Figure 3.10 this implementation is not a correct implementation of *Example*. This fact can be discovered by the test suite T derived with Autolink because, after performing the trace ab, *Imp* will send the output d' in place of d and, after performing the trace abc, *Imp* will send the output d' in place of d. Both of these situations will lead to a **fail**. The fact that *Imp* **passes** T does not hold is correctly discovered by the non-inclusion of the sets $P_{T,Imp}$ and T.

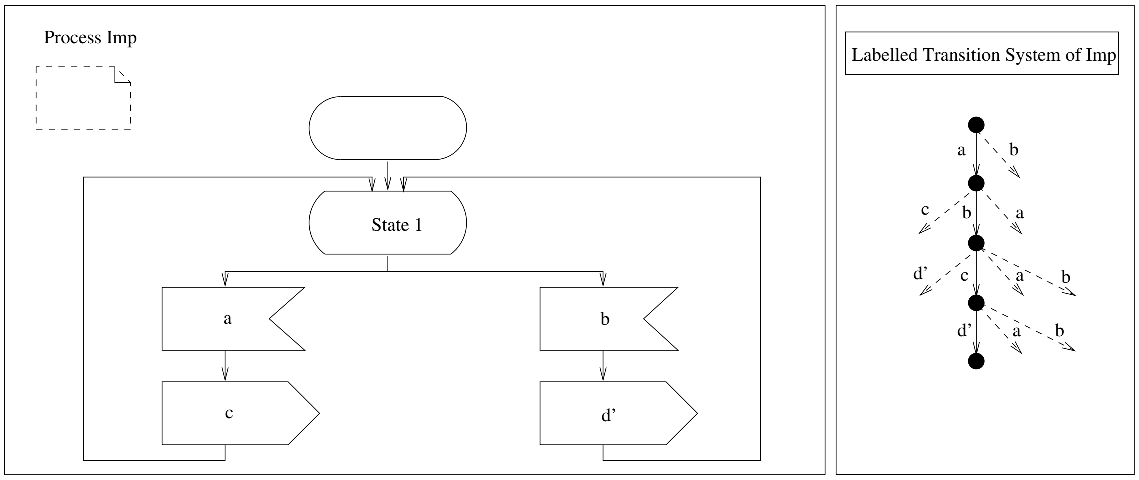


Figure 3.10: The *Imp* implementation.

Now we will define the test suite which is produced by Autolink.

Definition 3.4.3 Let Q be the labelled transition system of the SDL specification S. Let κ be the set of all predicates P defined by the options configuration of Autolink. Let ω be the set of all MSC which can be valid test purposes for S. Then: $T_{Auto} = \cup_{P\in\kappa, M\in\omega}$ **Autolink(**Q**,** $S(M)$**,** P**)** is the test suite formed by the set of test cases produced by the Autolink algorithm.

The key observation is that for every trace of the specification, one can build an MSC test purpose such that its set of traces contains the specification trace. By allowing full freedom in exploiting the variations of the options configuration, there will be at least one test case which contains this trace. So we can conclude that $traces(Q) \subseteq T_{Auto}$. Because of the presence of the δ produced by the Timeouts, T_{Auto} contains also traces which do not belong to $traces(Q)$ (the prefix trace which leads to δ (Timeout) is contained by $traces(Q)$, but the trace itself, because it ends with δ does not belong to $traces(Q)$).

Now we are ready to define a conformance relation. In order to deal with predicates P, each of which represents a specific setting of the options configuration, we take the following point of view: the tester is allowed to take full freedom in exploiting the variations of the options configuration. If there exists at least one setting by which a certain error can be detected, then this is added to the algorithm error detection capability as represented by the conformance relation.

Definition 3.4.4 Let *Spec* be the labelled transition system of the SDL specification S and *Imp* be the labelled transition system of an implementation (*Imp* is an $\mathcal{IOTS}$, input complete in every state - we made before this observation). Let T_{Auto} be the set of the test cases produced by the Autolink algorithm on *Spec*. Then: *Imp* **Auto** *Spec* $\Leftrightarrow$ *Imp* **passes** T_{Auto}.

It is quite easy to prove that Autolink generates sound tests and that it is exhaustive with respect to the **Auto** conformance relation. The exhaustiveness is a result of the definition itself and the soundness is easily proved because any test case is part of the set of all the test cases T_{Auto}.

3.4.3 Classification of Autolink

Theorem 3.4.5 gives the translation of the **Auto** conformance relation in *ioco* terms.

Theorem 3.4.5 *Let Imp be an implementation and let Spec be a specification. Let* $F_{Auto} = \{t \in traces(Spec) \mid \text{out}(\,Spec\ \textbf{after}\ t) \neq \{\delta\}\}$ *be the subset of traces from* $traces(Spec)$ *with the property that every trace from this subset makes the specification produce an output.*

1. $\forall$ *Spec, Imp: Imp* **Auto** *Spec* $\Leftrightarrow$ *Imp* $\textbf{ioco}_{F_{Auto}}$ *Spec;*
2. $\forall$ *Spec, Imp: Imp* **ioconf** *Spec* $\Rightarrow$ *Imp* **Auto** *Spec;*
3. $\exists$ *Spec, Imp*: $\neg$(*Imp* **ioconf** *Spec*) $\wedge$ *Imp* **Auto** *Spec;*

Proof:

1. According to Definition 3.4.4:

   $$Imp\ \textbf{Auto}\ Spec \Leftrightarrow Imp\ \textbf{passes}\ T_{Auto}$$

   The set P_T (Definition 3.4.2) is formed by all the traces of the specification which end with an output. Then:

   $$P_T = F_{Auto} = \{t \in traces(Spec) \mid out(Spec\ \textbf{after}\ t) \neq \{\delta\}\}$$

   The set $P_{T_{Auto},Imp}$ is:

   $$P_{T_{Auto},Imp} = \{t'o \mid t' \in F_{Auto}, o \in out(Imp\ \textbf{after}\ t')\}$$

   For having *Imp* **passes** T_{Auto}, the relation which should hold (see Definition 3.4.2) is:

   $$P_{T_{Auto},Imp} \subseteq T_{Auto}$$

   This means that:

   $$\forall t \in P_{T_{Auto},Imp} : t \in T_{Auto}$$

   Because $t = t'o$ with $t' \in F_{Auto}$ and $o \in out(Imp\ \textbf{after}\ t')$ the relation becomes:

   $$\forall t'o \in P_{T_{Auto},Imp} : t'o \in T_{Auto}$$

Now $t' \in traces(Spec)$ because $t' \in F_{Auto}$ and $F_{Auto} \subseteq traces(Spec)$. For having $t'o \in T_{Auto}$ it is necessary that $o \in out(Spec\ \textbf{after}\ t')$. This means that:

$$\forall t' \in F_{Auto}, o \in out(Imp\ \textbf{after}\ t') : o \in out(Spec\ \textbf{after}\ t')$$

This means that:

$$\forall t' \in F_{Auto} : out(Imp\ \textbf{after}\ t') \subseteq out(Spec\ \textbf{after}\ t')$$

Which we recognize as the definition of *Imp* $\textbf{ioco}_{F_{Auto}}$ *Spec*. So we proved the point 1) of the theorem.

2. The point 2) of the theorem is now easily proved because **ioconf** has more discriminating power than $\textbf{ioco}_{F_{Auto}}$ $(F_{Auto} \subseteq traces(\text{Spec}))$.

3. For the point 3), let us consider the specification *Spec* and the implementation *Imp* from the Figure 3.11. The conformance relation **ioconf** is able to detect that after the trace a, *Imp* produces the output c which is not specified by *Spec*. Autolink can not check what output is produced after the trace a, because it looks at the outputs only in the moment when the specification produces an output (different from the null output); this is not the case for the trace a. So Autolink will let the error in *Imp* go undetected. □

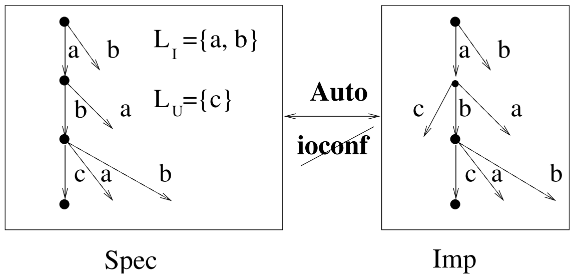


Figure 3.11: The relation between **ioconf** and **Auto**.

So we showed that **Auto** has less detection power then **ioconf**. And in practice, in order to arrive at its theoretical error detection capacity, an infinite number of MSCs must be created, which is impractical (typical methodologies are based on the interactive use of the simulator or validator to create MSCs; a similar remark applies to TGV; for TorX one needs infinitely many test cases, but these are generated automatically). Because [Tre96] shows that the **ioco** conformance relation has more discriminating power than **ioconf**, the Autolink algorithm has also less detection power than TorX and TGV. As a remark: there exists a version of TGV which works on SDL specifications (see Section 3.2); theoretically TorX can also work on LTS derived from SDL systems, although such a version does not exist in practice.

Below we will provide some discussion related to the comparison of Autolink with UIO and UIOv algorithms. We do not claim that we are exhaustive related to all the modifications and assumptions which should be made for Autolink to work on FSM. We just provide, at a high level, a guideline of how this can be done.

For comparing Autolink with UIO and UIOv methods we should adopt an FSM view, in the sense that a *null* output can be present in the output set. Below, we will sketch a way of modifying Autolink for working on FSM, under the assumption that a *null* output can be present in the output set. For such an output to exist, then the interface between the TTCN test (produced by Autolink) and IUT should be able to detect whether a waiting time is specified in the current moment of execution (by a *null* output of the TTCN test) and then interpret this waiting time as *null* and continue the execution of the test. Otherwise a Timeout will be generated. For making it possible to apply all the formalization from above, there should be two kinds of δ. The *null* output which is contained by the set of outputs is the first δ. The second δ, different only as notation from *null*, will be the one correspondent to a Timeout, i.e. an unspecified *null* output. In this way Autolink can generate test cases containing *null* outputs and also generate Timeouts. Under these considerations Autolink can work on LTSs derived from FSMs and a similar F_{Auto} can be constructed for LTSs derived from FSMs. Then based on the observation that $F_{UIO} \subseteq F_{UIOv} \subseteq F_{Auto}$, we can conclude that in principle Autolink is more powerful than the UIO and UIOv algorithms.

3.5 Concluding remarks

In this chapter we classified four known algorithms: TorX, TGV, Autolink (Telelogic/TAU) and UIO (UIOv) algorithms (Phact, Conformance Kit). The classification was made as a function of the conformance relation on which they are based, each conformance relation being expressed in terms of *ioco* theory. Also we consolidated the conformance foundation of Autolink by reconstructing an explicit conformance relation for it. This research treats only this criterion of classification (it looks at the error detection power of the algorithms); other criteria such as complexity or timing are out of the scope of this research.

From the theoretical analysis it resulted that TorX and TGV have the same detection power. Autolink has less detection power because it implements a less subtle relation than the first two (some situations exist in which the former two can detect an erroneous implementation and Autolink can not). We can also remark that for TGV and Autolink, in order to achieve their theoretical error detection capacity an infinite number of test purposes should be created, which is not always practical. For TorX one needs infinitely many test cases, but these are generated automatically which is also impractical because of resources and time limitations.

For comparing UIO and UIOv methods with Autolink, TGV and TorX one needs to assume that these tools are restricted to work on FSMs. Moreover, the restrictions implies different views regarding the presence or the absence of a null output in the output set for Autolink and *ioco* algorithms (TorX and TGV). Considering that such versions exist, it can be also concluded that UIO algorithms (Phact) have less detection power than Autolink, TGV and TorX. When the assumptions on which UIOv is based hold (connected and minimal FSMs, the number of the states of the IUT is less than or equal to the number of the states of the specification), the UIOv algorithms have the same detection power as the three algorithms restricted to work on FSMs; but, because these assumptions do not hold always in practice, we can conclude that the three algorithms are in general more powerful than the UIOv algorithms.

As we mentioned in the introductory section of this chapter, there was also a benchmarking experiment with these algorithms for test derivation. This experiment, complementary to the theory presented here, will be described in the next chapter of the thesis.

Chapter 4

Testing the Conference Protocol

4.1 Introduction

One of the subgoals of the CdR project is benchmarking existing tools, using a common case study: the Conference Protocol. The aim is to get insight in the strengths and weaknesses of the different approaches, to identify shortcomings in the used approaches and to identify comparison criteria, required computational effort and means to accommodate automation. Performance indicators for benchmarking include test preparation time, test execution time and coverage.

In this context we performed a case study on the *conference protocol* as described in [TPHT96]. The conference protocol is a simple but non-trivial example. Several implementations in C exist next to specifications in e.g. SDL, LOTOS, PROMELA and FSM. The conference protocol's main function is to provide a kind of *chat box service*. Users can register and un-register to conference chat boxes. Once registered, a user can have his messages broadcasted to his conference partners.

The conference protocol implementations were automatically tested in two ways: on-the-fly and batch-wise. The former addresses simultaneously test derivation and test execution, while the latter first derives a series of tests, which are executed after derivation. The batch testing experiment is based on the TAU tool set [AB98] using SDL, Phact using FSM and TGV using LOTOS. The on-the-fly testing experiment is based on TorX using LOTOS and PROMELA.

This experiment complements the theoretical comparison reported in Chapter 3. The comparison gave rise to three papers [BFdV$^+$99], [BRS$^+$00], [HFT00]. Our contribution in the whole effort was the testing experiment with Autolink on the SDL Conference Protocol specification. This includes the development of the SDL specification, developing MSC test purposes for the Conference Protocol, deriving test cases with Autolink and running test cases against mutants. We will present the experiment with Autolink on the Conference Protocol in detail. We will describe briefly the experiments with TorX, TGV and Phact and we will compare the results obtained.

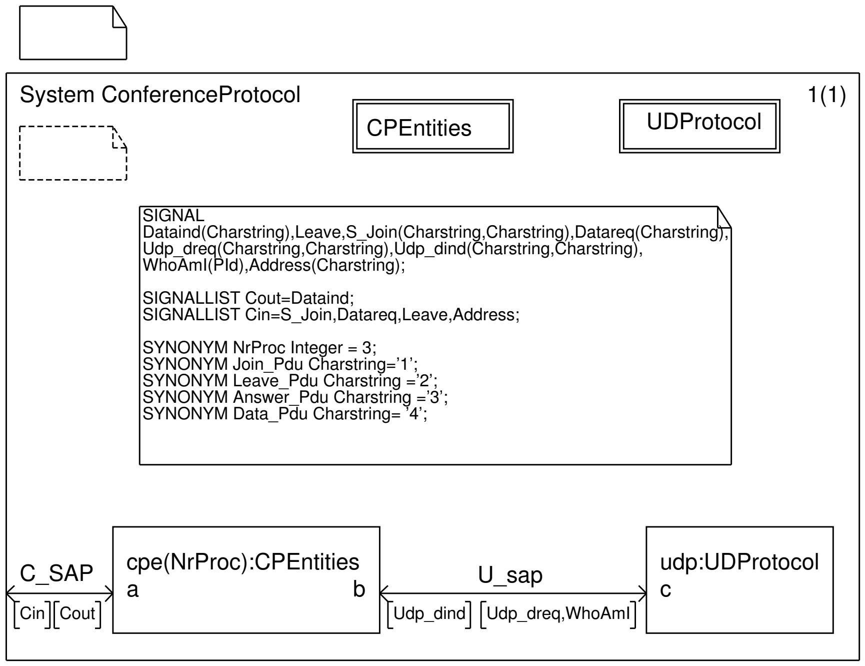
This chapter is structured as follows: Section 4.2 will explain the Conference Protocol specifications. The SDL specification will be presented in detail and the LOTOS, PROMELA and FSM specifications will be described briefly. We will give also an overview of the existing implementations in this section. Section 4.3 reports the test architecture used for the Conference Protocol case study and the validation of the specification. Section 4.4 describes the testing activity. Again, our work, viz. the testing experiment with Autolink, will be presented in more details, while the experiments with TorX, TGV and Phact will be given briefly. Section 4.5 presents the conclusions.

4.2 The Conference Protocol

The conference protocol described in this section is a simple protocol especially designed for studying and comparing test methodologies. A detailed description of this protocol can be found in [TPHT96]. Several implementations and specifications are available. The purpose of this section is to explain the conference protocol and to show some highlights from our SDL specification. This will be done in Section 4.2.1 and Section 4.2.2. Next to the SDL specification, the other specifications of the Conference Protocol, in LOTOS, PROMELA and FSM, are briefly presented in Section 4.2.3. Section 4.2.4 describes the implementations of the Conference Protocol.

4.2.1 Overall description

The purpose of the protocol is to connect a number of users to each other and to support them in exchanging messages within conferences. A conference is identified by a conference name and contains a number of users. There can be several conferences at the same time, but each user is engaged in at most one conference. A user can join a conference and leave a conference. This gives rise to the service primitives *S_Join* and *Leave*. Once engaged in a conference, a user can send a multicast message to all other users engaged in this conference, or receive a message from one of the other users. Thus we have two more service primitives, *Dataind* and *Datareq*. The primitives *Address* and *WhoAmI* serve for initialisation purposes and needs no be discussed here. Figure 4.1 shows the architecture of the Conference Protocol as an SDL diagram. The above mentioned service primitives occur as signals exchanged with the environment (representing the user) via the Conference Protocol Service Access Points (*C_SAP*).



```
SIGNAL
Dataind(Charstring),Leave,S_Join(Charstring,Charstring),Datareq(Charstring),
Udp_dreq(Charstring,Charstring),Udp_dind(Charstring,Charstring),
WhoAmI(PId),Address(Charstring);

SIGNALLIST Cout=Dataind;
SIGNALLIST Cin=S_Join,Datareq,Leave,Address;

SYNONYM NrProc Integer = 3;
SYNONYM Join_Pdu Charstring='1';
SYNONYM Leave_Pdu Charstring ='2';
SYNONYM Answer_Pdu Charstring ='3';
SYNONYM Data_Pdu Charstring= '4';
```

Figure 4.1: The Conference Protocol System.

The service provided by the Conference Protocol is not reliable. That means that messages may get lost, but never get corrupted. Also, messages may be delivered out of sequence. We will not discuss the treatment of error situations that can occur due to the loss of messages.

As shown in Figure 4.1, each user is supported by a dedicated Conference Protocol Entity (*CPE*). These CPEs implement the Conference Protocol by exchanging information via an underlying communication service. The Protocol Data Units exchanged between the CPEs are: *Join_Pdu*, *Leave_Pdu*, *Answer_Pdu*, and *Data_Pdu*. These messages are packed into frames and sent to or received from the underlying communication service. Inside a frame the types of the PDUs are identified by numbers from one to four: 1 for *Join_Pdu*, 2 for *Leave_Pdu*, 3 for *Answer_Pdu* and 4 for *Data_Pdu*. These numbers are represented as characters in the SDL specification (for example *'1'* stands for 1).

This underlying service is provided by the User Datagram Protocol (UDP, see [Com91]). This is a connectionless unreliable service, which means that messages may get lost and may be delivered out of sequence, but they never get corrupted, nor are they misdelivered.

The service primitives offered by the User Datagram Protocol are *Udp_dind* to receive a message from the UDP and *Udp_dreq* to send a message to the UDP. In the SDL specification from Figure 4.1 we use the type *Charstring* to pack Protocol Data Units of the Conference Protocol Entities into UDP service primitives.

At the end of this section we will give some explanation regarding the different types of boxes which occur on the SDL system diagram from Figure 4.1. For a detailed description of them we refer to [CCI92]. The box which is located outside of the main diagram containing the description of the SDL system, on top and to its left, represents the *package reference symbol*. The package reference contains references to packages with definitions that are to be included in the SDL system (see 'Reference in Package' and 'Reference in System' [CCI92]). We are using none of its functionalities in this diagram. Inside the main diagram and below the system name (*Conference Protocol*), the *additional heading symbol* is situated. It looks like a dashed package reference symbol. Some uses of it are: to define inheritance and formal parameters. We are not using its functionalities in this diagram. To the right of the system name there are two boxes (two rectangles each) which represent *block type reference symbols*. In this diagram they contain the types *CPEntities* and *UDProtocol*. These types are the types of the blocks which describe the behaviours of *CPE* and *UDP*. The blocks for *CPE* and *UDP* are defined by the boxes at the bottom of the diagram. In the central part there is the *text symbol*. Inside the text symbol, there can be defined signals (for example *Dataind*), signal lists (for example *Cout*), data type definitions (not shown in this diagram), etc. As we already mentioned, in the bottom there are two *block boxes* for *CPE*s and *UDP*.

4.2.2 Process description

The purpose of the Conference Protocol Entity is to maintain protocol information, such as the set of users engaged in the same conference, and to transmit data from and to the user via the underlying communication service. In this section we will only give a global description of the behaviour of a CPE. The main loop of a Protocol Entity is displayed in Figure 4.2. The purpose of the initialization procedure is to build certain data structures and to collect information about the configuration of the system. After initialization, the CPE switches between two main states, namely, *idle* (i.e. not engaged in a conference) and *engaged* (in a conference).

When the CPE is in state *idle*, it simply awaits a join message from the user, attributed with the user name and the name of the conference to join. This join message must be forwarded to all users. This is taken care of by procedure *TransAll*. The message transmitted consists of the type of the message (a *Join_Pdu*), the user name of the joining user and the conference name. Procedure *Append* is invoked in order to append an entry to the list of users engaged in this conference. Directly after this join message, the joining user is added as the only entry of this list.

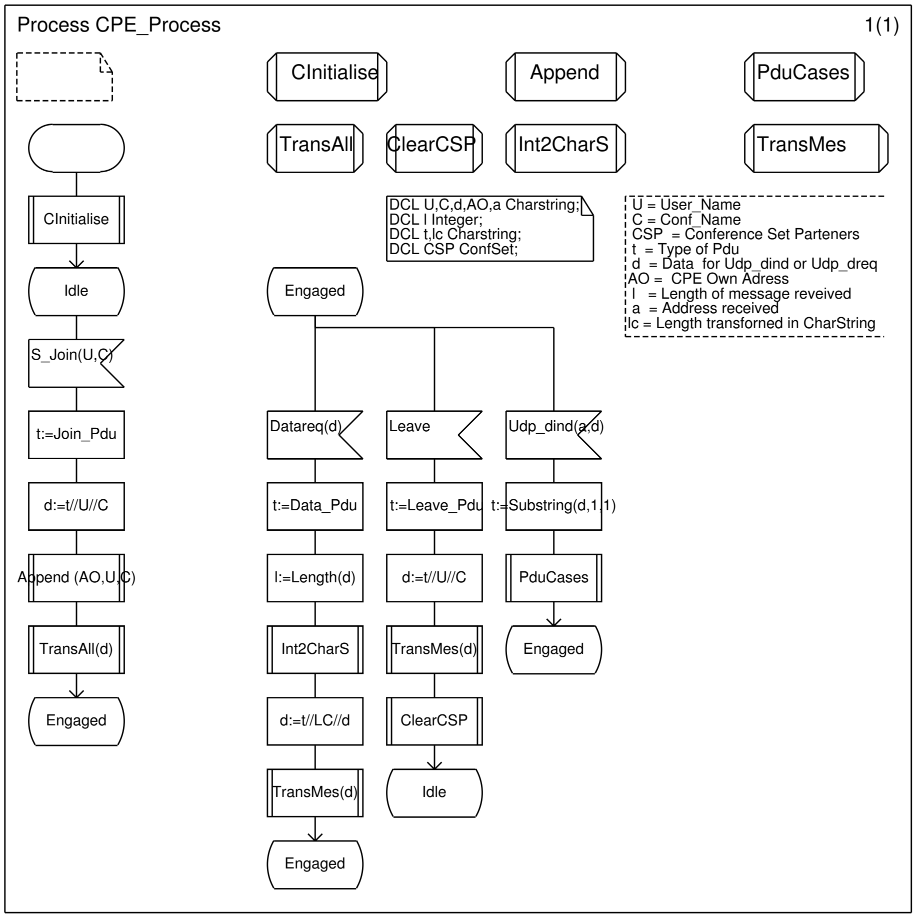


Figure 4.2: Behaviour of a Conference Protocol Entity.

Once in state *engaged*, the CPE is ready to accept several messages. Reception of a *Leave* message brings CPE back to the *idle* state. However, the CPE must first inform all other CPEs engaged in the same conference about this leaving. This is done by procedure *TransMes* sending a *Leave_Pdu*. The difference between *TransMes* and the aforementioned procedure *TransAll* is that the former sends a message to all users engaged in the same conference, while the latter sends a message to all users of the system. The set of all users of the system (the set of potential users) is determined once in the initialization phase, while the set of users engaged in the same conference (the set of conference partners) is updated after every join or leave message. Before entering the *idle* state, the list of users engaged in the same conference must be cleared (procedure *ClearCSP*).

In case the user wants to send a message to all participants of the conference, a *Datareq* is received by the CPE. This message is packed with some other information into a frame, which is sent to all participants of the conference. The extra information is the type of the PDU (*Data_Pdu*) and the length of the message, length which will be transformed into a charstring by the procedure *Int2CharS*.

Next, we consider the case that a message from the underlying service arrives (a *Udp_dind*). The data in this message contains the PDUs exchanged between the CPEs. There are four different PDUs: *Join_Pdu*, *Answer_Pdu*, *Data_Pdu*, and *Leave_Pdu*, which are all treated by procedure *PduCases*. The type of the PDUs is extracted from the first position of the message by the procedure *Substring*

Reception of a *Join_Pdu* with respect to the same conference results in an *Answer_Pdu* to the originator of the message. In this way, the CPE issuing the join message can collect information about all users currently subscribed to the conference.

Reception of an *Answer_Pdu* is considered a reply to the join of this CPE. The only consequence of this *Answer_Pdu* is an update of the list of participants.

Reception of a *Leave_Pdu* also only affects the list of participants.

An incoming *Data_Pdu* is simply passed on to the user.

The behavioural description of the underlying communication service, the User Datagram Protocol is not given here. Its behaviour is simply that of a lossy, unordered queue.

Finally, we remark that the above specification of the Conference Protocol is not the only one that we have produced. As will be explained in Section 4.3, we have exercised with different (small) variations of the above SDL specification.

4.2.3 LOTOS, PROMELA and FSM specifications

The LOTOS specification (see [BFdV$^+$99]) was mainly taken from [TPHT96]. The core of the specification is the (state-oriented) description of the conference protocol entity behaviour, which closely follows the rules sketched above. The CPE behaviour is parameterized with the set of potential conference partners and its *C_SAP* and *U_SAP* addresses, and is constrained by the local behaviour at *C_SAP* and *U_SAP*. The instantiation of the CPE with concrete values for these parameters is part of the specification.

In the PROMELA specification (see [BFdV$^+$99]), the communication between conference partners has been modelled by a set of processes, one for each potential receiver, to ‘allow’ all possible interleavings between the several sendings of multicasted PDUs. Instantiating the specification with three potential conference users, a PROMELA model for testing is generated which consists of 122 states and 5 processes. For model checking and simulation purposes the user not only needs to provide the behaviour of the system itself, but also the behaviour of the system environment. For testing, this is not required. Only some additional channels have to be marked as observable, viz. the ones needed to check for in the test derivation algorithm.

A detailed description of the FSM model of the Conference Protocol can be found in [HFT00]. We will outline here some of its elements. An EFSM (Extended Finite State Machine) description of the Conference Protocol was written first. The EFSM was then converted in an FSM. Because the FSM is considered by Phact to be a Mealy machine (see Section 3.3), there were some restrictions on the way in which the Conference Protocol was modeled. For example, the alternation between inputs and outputs is always required for an FSM. This implies that a sequence of multiple inputs with delayed outputs is not considered, and hence not tested. Another restriction is that an FSM is not allowed to have parameterized inputs and outputs. Therefore the number of conferences and the number of active partners had to be fixed in the ESFM Conference Protocol specification to 2 and 3, respectively. These are some restrictions which affected the design of the EFSM Conference Protocol. The main effect of the restrictions is that some behaviours of the Conference Protocol could not be fully modeled and therefore they could not be completely tested.

4.2.4 Conference Protocol Implementations

A conference protocol implementation is implemented on SUN SPARC workstations using a UNIX-like (SOLARIS) operating system, and it is programmed using the ANSI-C programming language. Furthermore, only standard UNIX inter-process and inter-machine communication facilities, such as uni-directional pipes and sockets have been used.

For the benchmarking experiment 28 different conference protocol implementations were developed. One of these conference protocol implementations is correct (at least, to the best of our knowledge), whereas in 27 of these implementations a single error was injected deliberately. The erroneous implementations can be categorized in three different groups: *No outputs*, *No internal checks* and *No internal updates*. The group *No outputs* contains implementations that fail to send output when they are required to do so. The mutants from this group are denoted (using the mutant numbers of the internal identification scheme, used also in [HFT00]) as 100, 111, 384, 548, 674 and 687. The group *No internal checks* contains implementations that do not check whether the implementations are allowed to participate in the same conference according to the set of potential conference partners and the set of conference partners. The mutants of this group are denoted as 293, 398, 444 and 666. The group *No internal updates* contains implementations that do not correctly administrate the set of conference partners. The mutants of this group are denoted as 214, 247, 276, 289, 294, 332, 345, 348, 358, 462, 467, 738, 749, 777, 836, 856 and 945.

4.3 Validation and testing

In this section we will explain the test architecture which was used for testing the Conference Protocol. This will be done in Section 4.3.1. Section 4.3.2 deals with the validation of the SDL specification.

4.3.1 The test architecture

In Section 4.2 we described the general architecture of the Conference Protocol. The general architecture is also represented in Figure 4.3. For testing certain restrictions were adopted with respect to the number of CPEs and the reliability of the UDP service. For this reason the SDL specification used for testing was a simplified version of the general SDL specification which formalizes the Conference Protocol. We present the simplified architecture in Figure 4.5. Figure 4.4 shows an intermediate architecture between the general one, from Figure 4.3, and the simplified one, from Figure 4.5. This architecture was not used in practice.

The general architecture of the Conference Protocol with a complete number of CPEs and an underlying communication service is shown in Figure 4.3.

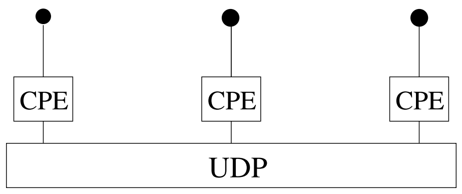


Figure 4.3: The architecture used for the internal check of errors.

The architecture is formed from three CPEs and one (unreliable) UDP. These are specified as explained in Section 4.2. The number of the CPEs is three because this is the minimum number required to check various aspects of the Conference Protocol. This number is also used for the Distributed Single Layer test architecture as discussed below (see Figure 4.5).

The architecture from Figure 4.4 is used for conformance testing of a CPE. The CPE is tested in isolation, in an OSI IS-9646 style (see Chapter 1). It simply consists of a single CPE embedded in a test environment. It is called an Ideal test architecture, since it supports testing of the CPE in isolation, without being disturbed by the UDP service.

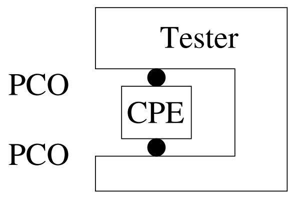


Figure 4.4: The ideal architecture.

This ideal architecture is very suited for automatic test generation. But regrettably, it is an unrealistic architecture. In all practical test situations, the UDP can not be excluded. Therefore this architecture could not be used in practice.

The third architecture does assume the presence of an implementation of the UDP service (see Figure 4.5). It consists of one single CPE and one UDP service. It is called a Distributed Single Layer test architecture and it is a variant of the Distributed Test Method architecture of OSI IS-9646 (see Chapter 1).

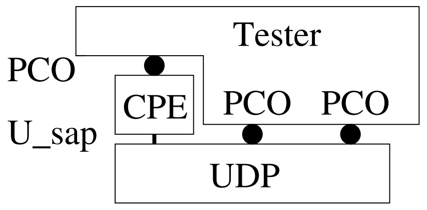


Figure 4.5: The Distributed Single Layer test architecture.

For the purpose of conformance test generation the UDP is assumed to be reliable. This is not really true, but it makes test generation and benchmarking easier. In this architecture, the tester must not only provide input for the CPE under test; it should also emulate two peer CPEs. Benchmarking of testing tools and methodologies in the Côte-de-Resyste project is performed on the basis of this third architecture from Figure 4.5.

The main differences between the SDL specifications, respectively the general one from Figure 4.3 and the simplified version from Figure 4.5, are the following:

- In the general SDL specification we could use all SDL constructs available for the complete architecture. However, for test derivation a connection to the TTCN language had to be established, which restricted the use of certain SDL constructs. The SDL type PId, e.g., used for constructing addresses, had to be converted into a different type.
- As discussed before, the complete architecture contains a specification of an unreliable UDP, while the Distributed Single Layer test architecture contains a reliable UDP.
- In the general architecture the number of CPEs can be considered as a parameter of the system. For conformance testing, the number had to be fixed: one CPE was specified and other two CPEs were assumed to participate in the network.

4.3.2 Validation

While specifying the Conference Protocol system in SDL, an important issue was correctness of the obtained specification. Since complete verification was not the central issue in this testing project, we did not pursue complete formal verification of the correctness of the protocol. In order to improve confidence and to detect internal errors, we used the simulator and the validator of the TAU tool set. Both tools were applied to the general system specified in Figure 4.3. The simplified SDL version was also checked for finding internal errors. Because this check was not so laborious as the one of the general architecture, we will refer only to the checking of the general one. Moreover once we were confident that the general system is error free, we could also be more or less confident that the simplified version is error free.

The system was first simulated. A simple MSC, called Simulation Trace, derived from such a simulation is shown in Figure 4.6. This figure shows three CPEs communicating with their users (i.e. the environment). All users are represented by the same instance. The communication between the CPEs (by means of the underlying communication service) is not shown in this figure. The scenario represented in the MSC describes users *Mark*, *John* and *Alex* joining conference *Conf*, after which *John* leaves the conference again. Then *Mark* sends a message to the conference, which is received by *Alex* only.

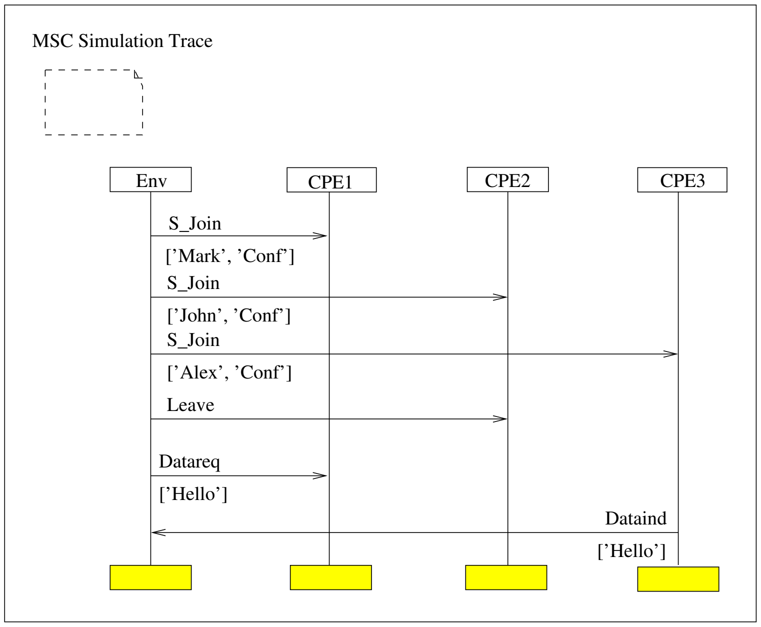


Figure 4.6: One MSC produced by a simulation.

The validator was used to prove absence of deadlock and to check several invariants, such as the following two.

- the list of users contains only the names of the users which participate in the same conference;
- the CPE produces only messages of type *Join_Pdu*, *Leave_Pdu, Answer_Pdu,* or *Data_Pdu*.

The other specifications in LOTOS, PROMELA and FSM were also checked for the detection of internal errors. We refer to [BFdV+99], [BRS+00], [HFT00] for more information regarding the checking activity for these specifications.

4.4 Testing Activities

This section describes our testing activities. The 27 erroneous mutants were tested without knowledge of which errors had been introduced in these mutants. The SDL testing with Autolink is presented in Section 4.4.1. This activity is our main contribution in the benchmarking experiment. The test experiments performed with TorX, TGV and Phact are briefly presented in Section 4.4.2.

4.4.1 Autolink testing

We used the TAUtool kit for batch derivation and execution of test suites in TTCN from SDL [AB98]. The TAU Autolink test derivation tool generated TTCN test suites from the SDL specification, guided by MSCs that were derived manually from the SDL specification using TAU's SDL simulator. A fragment of such an MSC is given in Figure 4.7.

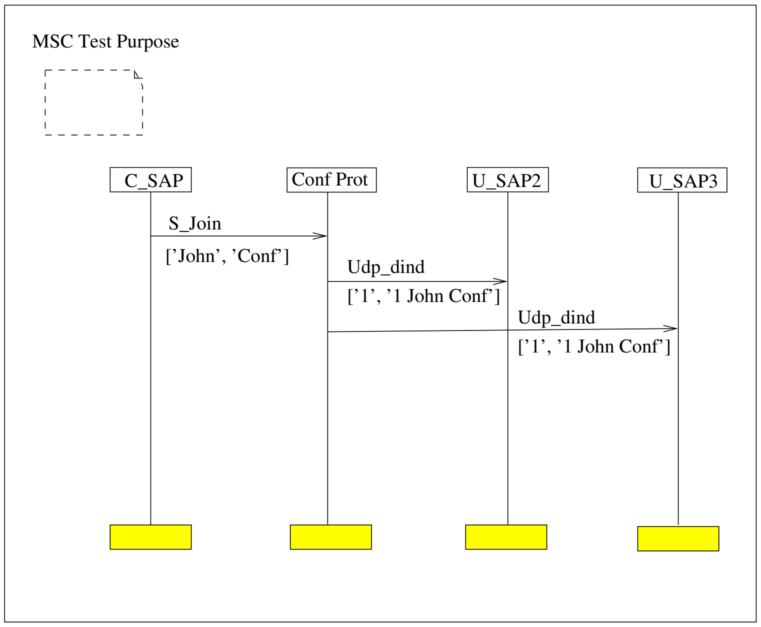
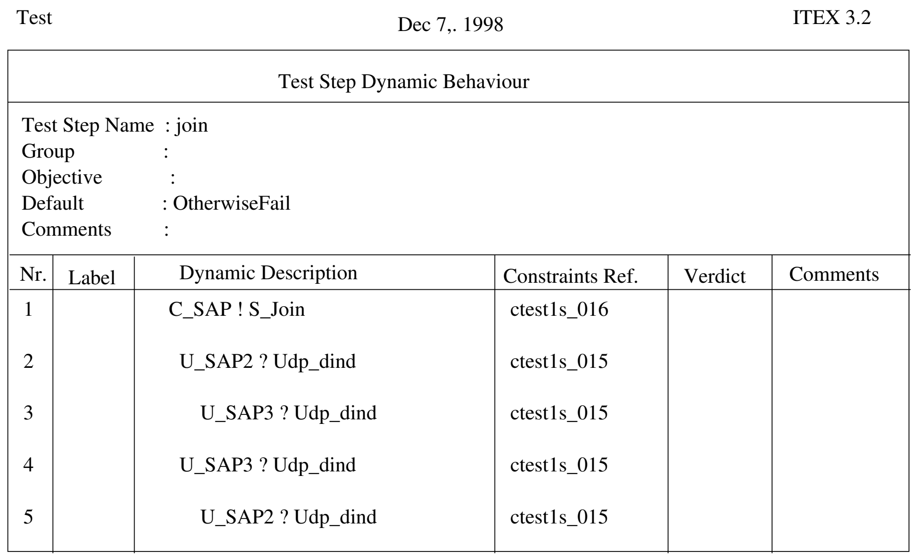


Figure 4.7: Fragment of an MSC describing a test purpose.

The SDL and MSC specifications are used by the Autolink tool for producing the constraint and dynamic parts of a TTCN suite. See Figure 4.8 for an example. In order to complete the test suite, a link executable is needed, which is derived from the SDL specification by the Link tool. This executable contains information which is necessary to generate type declarations (e.g. for the PCOs).

Before running the derived TTCN test suite against our implementations, we have run it against the original SDL specification, to validate the TTCN test suite, by running the SDL simulator and Tau's TTCN simulator in connection. This uncovered some problems in Autolink. Another problem encountered was that for some MSCs no TTCN suites could be derived, although the MSC could be successfully verified (this depends on the memory of our computers). Some of the above mentioned problems may have been solved already in recent releases of the software involved.

The number of test cases derived equals 15. The time to build an MSC by means of simulation is 3 minutes for an MSC with 7 events which includes the time to split it into parts (two parts in this case). The division of an MSC test purpose in parts corresponds to the division of a test case in test



Test | Dec 7,. 1998 | ITEX 3.2

Test Step Dynamic Behaviour

Test Step Name : join
Group :
Objective :
Default : OtherwiseFail
Comments :

Nr.	Label	Dynamic Description	Constraints Ref.	Verdict	Comments
1		C_SAP ! S_Join	ctest1s_016		
2		U_SAP2 ? Udp_dind	ctest1s_015		
3		U_SAP3 ? Udp_dind	ctest1s_015		
4		U_SAP3 ? Udp_dind	ctest1s_015		
5		U_SAP2 ? Udp_dind	ctest1s_015		

Figure 4.8: The corresponding TTCN code for the previous MSC.

steps (see Chapter 1). The 3 minutes represent the shortest time. The longest time was 45 minutes for 44 events (15 parts). The average time was 12 minutes for an average of 20 events and 5 parts. The test derivation time consists of the abovementioned times plus the time of the Autolink step generation which was less than 8 seconds. The preceding splitting turned out essential for most cases (e.g. 14 minutes without splitting becomes 7 seconds with splitting). Two test cases which involved MSC with many events could not be derived – not even after splitting.

We will sketch our informal strategy for defining test purposes in the next few lines. Most of the test purposes are concerned with a single conference. Various arbitrary interleavings of join actions, data transfer and leave actions give rise to one test purpose each. The other test purposes check the absence of interference between two simultaneous conferences (for 3 users it makes no sense to have more than 2 conferences).

The test execution time, running the TTCN which was derived in a batch-wise way against the implementation, took from 2 to 5 seconds per test.

The detection of errors was done by repeating all the remaining 13 test cases for the 27 mutants and the correct implementation. The correct implementation gets a **pass** verdict (the same happened for TorX, TGV and Phact). Six **fail** verdicts were obtained for the mutants 289, 294, 332, 467, 749 and 945. The 15 mutants 111, 214, 247, 276, 293, 345, 348, 384, 444, 462, 548, 674, 687, 738 and 836 were given **inconclusive** verdicts that were the effect of a Timeout. Six mutants (100, 358, 398, 666, 777, and 856) went undetected, although some of them could have been found by a larger test suite, but this costed at least several minutes per case. After analyzing the execution traces which led to Timeouts, the Timeouts for 11 mutants represented discoveries of real errors. For four mutants (214, 293, 444 and 836) the Timeouts could be justified by the behaviours of the SDL Conference Protocol specification and for this reason these Timeouts did not represent discoveries of errors. We will explain the Timeout analysis below.

The Timeout analysis took about 8 hours. The execution traces which led to Timeouts were short, in average with a length below 13. There were only three traces with a length above 13 and below 21. Because the traces were short, we could analyze these traces quite easily. After the analysis, the Timeouts revealed 11 errors in mutants. The Timeouts for four mutants could be justified by behaviours of the SDL specification. We will exemplify how the analysis was done for one discovery of an error, respectively for mutant 111, and for one Timeout which was justified by the specification, respectively for mutant 836. In a similar style the analysis for the rest of Timeouts was done.

The execution trace which led to a Timeout which revealed the error of mutant 111 is represented in Figure 4.9. The instances of the MSC represent: the Conference Protocol (mutant 111) and the

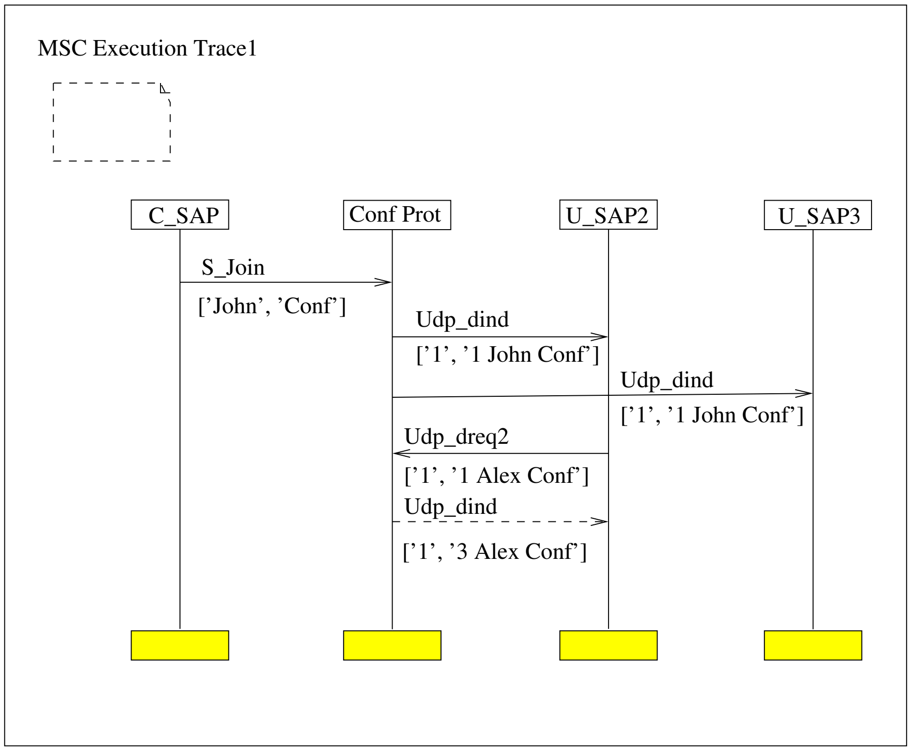


Figure 4.9: The execution trace for the Timeout of mutant 111.

PCOs (channels) *C_SAP*, *U_SAP2*, *U_SAP3* through which the mutant communicates with the environment. Sometime we will mention also *U_SAP* which connects the CPE under test with the UDP. The temporal ordering of the signals of the execution trace, as shown in the MSC of the Figure 4.9, is the following:

1. the user *John* engaged in the conference *Conf* by sending the signal *S_Join('John', 'Conf')* via channel *C_SAP* ; he is the first participant in this conference;

2. the assumed *CPE2* is announced that a user is engaged in the conference *Conf* by a *Join_Pdu* which is mapped in the output *Udp_dind('1', '1 John Conf')* sent via channel *U_SAP2*; we remind that the first parameter *'1'* of *Udp_dind* represents the address of the sending *CPE* and the substring *'1'* from the second parameter of *Udp_dind*, respectively *'1 John Conf'*, represents the type of PDU, *Join_Pdu*;

3. the assumed *CPE3* is announced that a user is engaged in the conference *Conf* by a *Join_Pdu* which is mapped to the output *Udp_dind('1', '1 John Conf')* sent via channel *U_SAP3*; this completes the registration of *John* as a participant of *Conf*;

4. the assumed *CPE2* sends a *Join_Pdu* to *CPE1* announcing that user *Alex* is engaged in the same conference *Conf* too (the other broadcast messages of *Alex*, e.q. *U_SAP3*, are suppressed here); this *Join_Pdu* is mapped in the input *Udp_dreq2('1', '1 Alex Conf')* sent via channel *U_SAP2*;

5. an *Answer_Pdu* mapped in *Udp_dind('1', '1 John Conf')* is expected to be received via *U_SAP2*, but no output occurs and the Timeout is produced instead.

We checked the transition between global states (see Section 3.4) of the SDL Conference Protocol specification to find the terminal global states to which this trace leads. If none of the terminal global states contains only inputs, in other words if each of them has the output *Answer_Pdu* as possible transition, then the null output is unspecified and the Timeout produced by this execution trace represents an error. The transitions between the global states of the SDL Conference Protocol specification are represented in Figure 4.10.

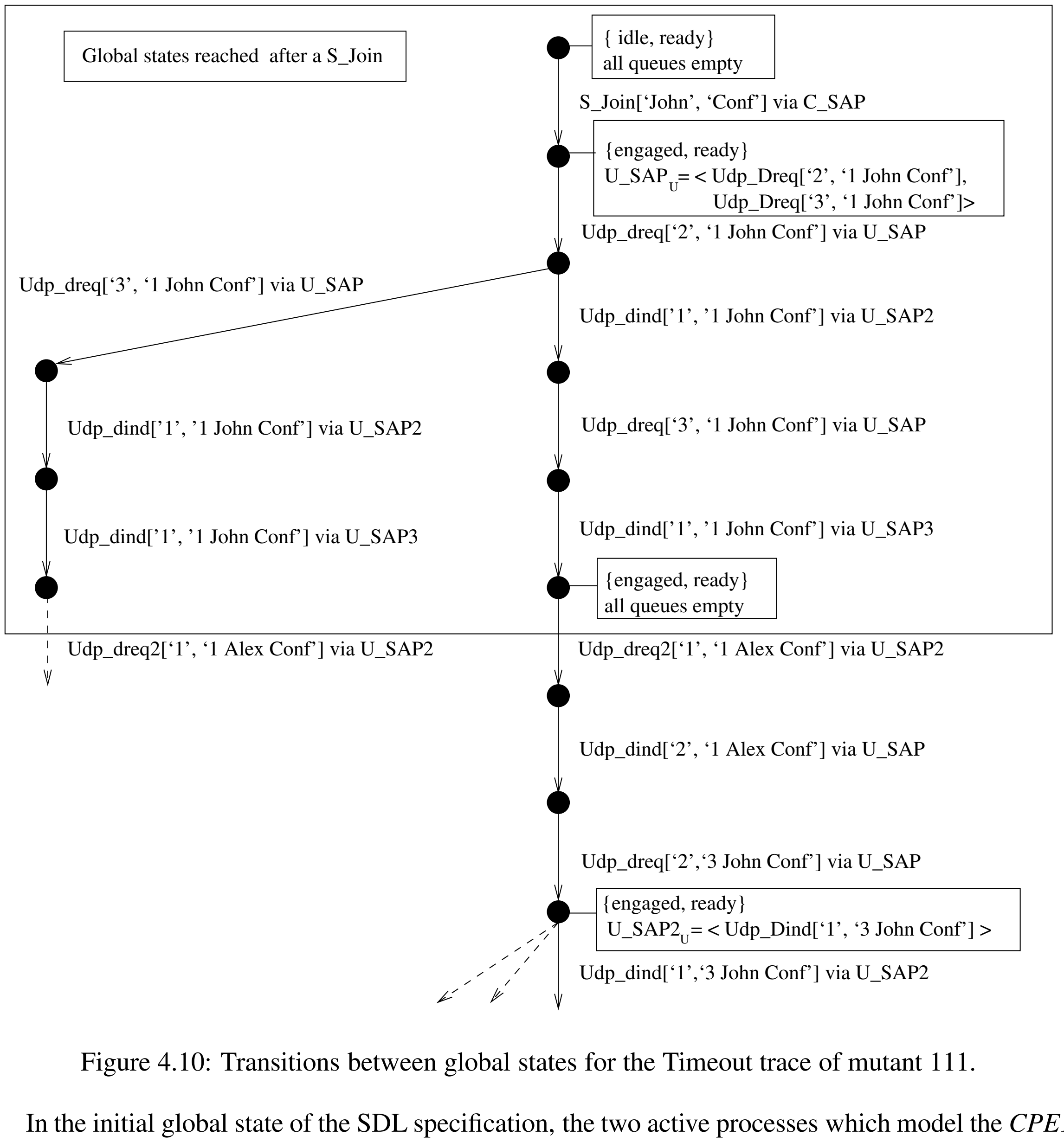

Global states reached after a S_Join
{ idle, ready}
all queues empty
S_Join['John', 'Conf'] via C_SAP
{engaged, ready}
U_SAP_U= < Udp_Dreq['2', '1 John Conf'],
Udp_Dreq['3', '1 John Conf']>
Udp_dreq['2', '1 John Conf'] via U_SAP
Udp_dreq['3', '1 John Conf'] via U_SAP
Udp_dind['1', '1 John Conf'] via U_SAP2
Udp_dind['1', '1 John Conf'] via U_SAP2
Udp_dreq['3', '1 John Conf'] via U_SAP
Udp_dind['1', '1 John Conf'] via U_SAP3
Udp_dind['1', '1 John Conf'] via U_SAP3
{engaged, ready}
all queues empty
Udp_dreq2['1', '1 Alex Conf'] via U_SAP2
Udp_dreq2['1', '1 Alex Conf'] via U_SAP2
Udp_dind['2', '1 Alex Conf'] via U_SAP
Udp_dreq['2','3 John Conf'] via U_SAP
{engaged, ready}
U_SAP2_U= < Udp_Dind['1', '3 John Conf'] >
Udp_dind['1','3 John Conf'] via U_SAP2

Figure 4.10: Transitions between global states for the Timeout trace of mutant 111.

In the initial global state of the SDL specification, the two active processes which model the *CPE1* and *UDP* are in the states *idle* and *ready*. Each channel has two queues associated: one for outputs and one for inputs. The convention is that the subscript *I* is added to the name of the channel for denoting the queue for inputs and the subscript *U* is added to the name of channel for denoting the queue for outputs. For example, for channel *C_SAP* the queue for inputs is denoted as C_SAP_I and the queue for outputs is denoted as C_SAP_U. All the queues of the channels are empty in the initial global state. Receiving *S_Join('John', 'Conf')* from queue C_SAP_I makes the SDL process of *CPE1* to go from state *idle* to state *engaged* while puting the outputs *Udp_dreq('2', '1 John Conf')* and *Udp_dreq('3','1 John Conf')* in the queue U_SAP_U of channel *U_SAP*. Each of the *Udp_dreq* outputs is a mapping of a *Join_Pdu* which announces an assumed *CPE* partner, *CPE2* or *CPE3*, that user *John* engaged in conference *Conf*. The consumption of these two *Udp_dreq* outputs represent internal transitions of the SDL system, transitions which are not visible for the environment. In the new global state reached, the set of active states is {*engaged*, *ready*}; the output queue U_SAP_U contains the two *Udp_dreq* and the rest of the queues are empty. The outputs contained by U_SAP_U are consumed in the same order in which they arrived in the queue: first *Udp_dreq('2', '1 John Conf')* and after that *Udp_dreq('3', '1 John Conf')*. Now the following sequences of transitions can be performed such that first the output *Udp_dind('1','1 John Conf')* via channel *U_SAP2* and then *Udp_dind('1','1 John Conf')* via channel *U_SAP3* are visible to the environment:

1. *Udp_dreq('2','1 John Conf')* from U_SAP_U, *Udp_dind('1','1 John Conf')* from U_SAP2_U, *Udp_dreq('3','1 John Conf')* from U_SAP_U, *Udp_dind('1','1 John Conf')* from U_SAP3_U;

2. *Udp_dreq('2','1 John Conf')* via U_SAP_U, *Udp_dreq('3','1 John Conf')* from U_SAP_U, *Udp_dind('1','1 John Conf')* from U_SAP2_U, *Udp_dind('1','1 John Conf')* from U_SAP3_U.

Each of these transitions leads to a global state (in total 2) in which the set of active states is {*engaged*, *ready*} and all the queues are empty. Now the input *Udp_dreq2('1','1 Alex Conf')* is sent from the environment via channel *U_SAP2* (see the execution trace from Figure 4.9). This input which is a mapping of a *Join_Pdu* causes the following chain of transitions between the global states of the SDL specification: 1) the starting point is a global state (one of the two which can be reached after *S_Join*) which has the set of active states {*engaged*, *ready*} and all the queues empty; 2) the sequence of transitions is: *Udp_dreq2('1','1 Alex Conf')* from U_SAP2_I, *Udp_dind('2','1 Alex Conf')* from U_SAP_I, *Udp_dreq('2','3 John Conf')* from U_SAP_U; 3) an ending state which has the set of active states {*engaged*, *ready*}, the output queue U_SAP2_U storing the output *Udp_dind('1', '3 John Conf')* which is a mapping of an *Answer_Pdu* and the rest of the queues empty. Each of the 2 ending states which can be reached has as possible transition the output *Udp_dind('1','3 John Conf')* which is present in the queue U_SAP2_U. Therefore a null output is not specified and for this reason the Timeout in this case represents an error.

Now we will analyse the execution trace which led to the Timeout for mutant 836. The situation is captured in the MSC from Figure 4.11.

The execution trace is longer than the trace represented in this figure. For simplifying the explanation we cut the beginning part which ends into a *Leave*. The *Leave* can be thought as being a reset signal which brings the system in the initial state. Therefore, without loss of generality, we could cut the initial part and still preserve the situation which led to the Timeout. The execution trace has the signals ordered in the following way:

1. the user *John* engaged in the conference *Conf* by sending the signal *S_Join('John', 'Conf')* via channel *C_SAP*;

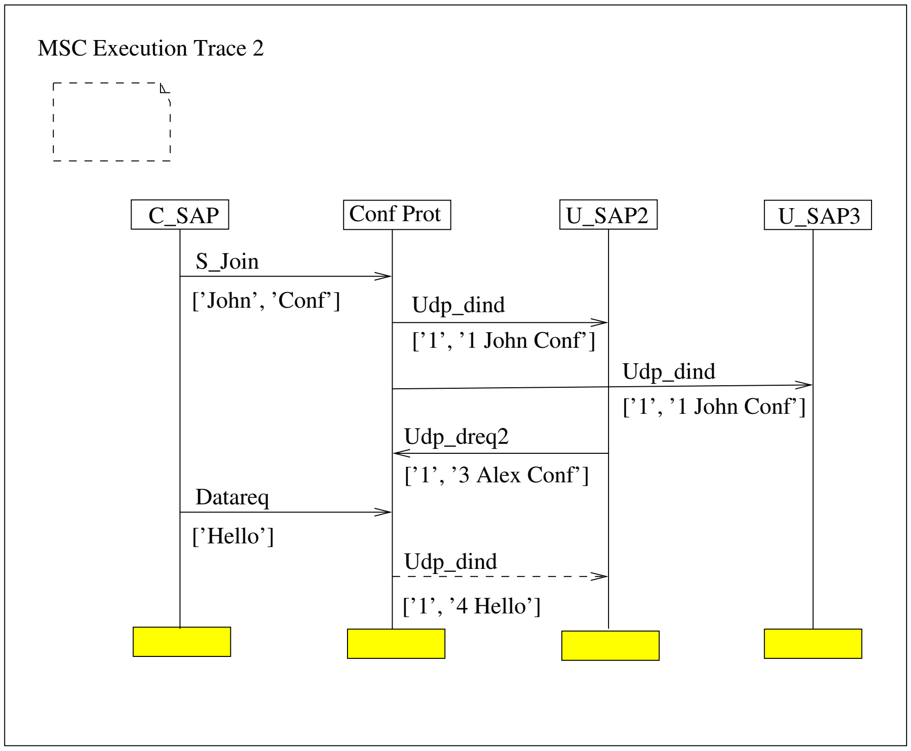


Figure 4.11: The execution trace for the Timeout of mutant 836.

2. the assumed *CPE2* is announced that a user is engaged in the conference *Conf* by a *Join_Pdu* which is mapped to the output *Udp_dind('1', '1 John Conf')* sent via channel *U_SAP2*;

3. the assumed *CPE3* is announced that a user is engaged in the conference *Conf* by a *Join_Pdu* which is mapped to the output *Udp_dind('1', '1 John Conf')* sent via channel *U_SAP3*;

4. the assumed *CPE2* sends an *Answer_Pdu* to *CPE1* announcing that user *Alex* is engaged in the same conference *Conf*; this *Answer_Pdu* is mapped into the input *Udp_dreq2('1', '3 Alex Conf')* sent via channel *U_SAP2*;

5. input *Datareq('Hello')* is sent via channel *C_SAP* to *CPE1*;

6. output *Data_Pdu* mapped to *Udp_dind('1', '4 Hello')* is expected to be received via *U_SAP2*, but none output occurs and the Timeout is produced.

There is a tricky point in this MSC situation. First we remind that an *Answer_Pdu* (from step 4)) causes no response from the CPE which received it. Now the tricky thing is that the two inputs (from step 4) and 5)) are sent one after another, without outputs between them. Completely performing *Answer_Pdu* and the internal transition associated with it before *Datareq* justifies the expectation of a *Data_Pdu* (from step 5)). But, if the *Answer_Pdu* is not performed completely before *Datareq* (*Datareq* is consumed immediately after the *Answer_Pdu* and before the internal transition associated with *Answer_Pdu*) then this will lead to a global state in which no output is expected (it contains only inputs as transitions). This global state, using the *ioco* terminology, is a quiescent state. In this quiescent state, a null output can be observed. Therefore the null output can be produced by the

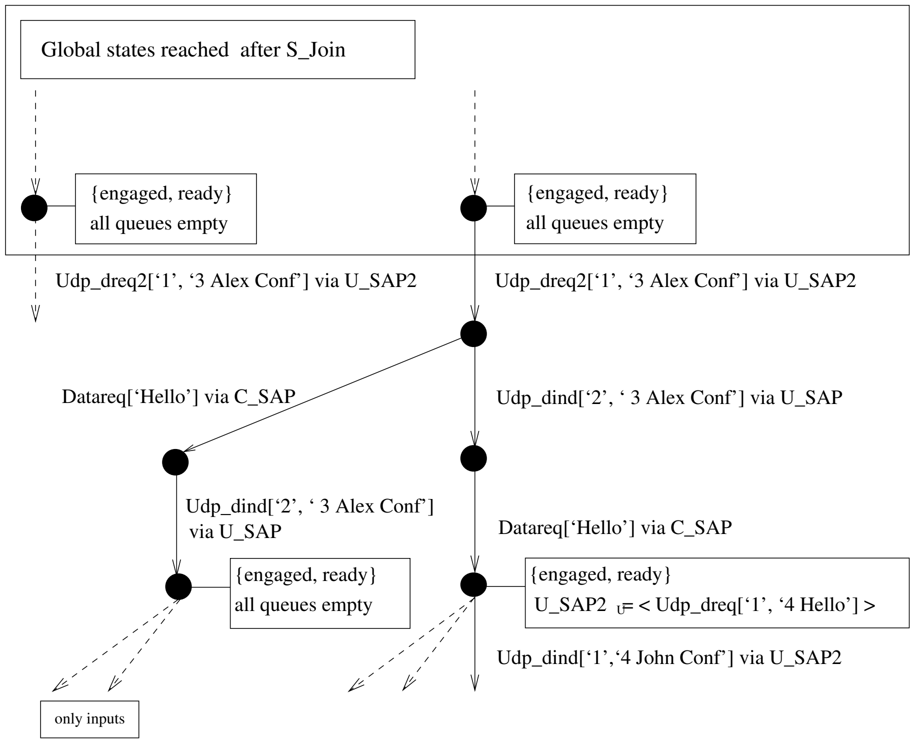

Global states reached after S_Join
{engaged, ready}
all queues empty
{engaged, ready}
all queues empty
Udp_dreq2['1', '3 Alex Conf'] via U_SAP2
Udp_dreq2['1', '3 Alex Conf'] via U_SAP2
Datareq['Hello'] via C_SAP
Udp_dind['2', ' 3 Alex Conf'] via U_SAP
Udp_dind['2', ' 3 Alex Conf'] via U_SAP
Datareq['Hello'] via C_SAP
{engaged, ready}
all queues empty
{engaged, ready}
U_SAP2 U= < Udp_dreq['1', '4 Hello'] >
Udp_dind['1','4 John Conf'] via U_SAP2
only inputs

Figure 4.12: Transitions between global states for the Timeout trace of mutant 836.

SDL specification after the execution trace from Figure 4.11 and, consequently, a Timeout event is specified.

In Figure 4.12 we showed the transitions between the global states of the SDL Conference Protocol specification for the MSC of Figure 4.11. We did not represent in this figure the transitions for *S_Join* (which includes the signals from step 1), 2) and 3)), because this is explained in Figure 4.10. In Figure 4.12 we represented only the transitions between global states for the sequence *Answer_Pdu*, *Datareq* (from step 4) and 5)). The starting point is one of the global states which can be reached after performing *S_Join*. In such a state, the set of active states is {*engaged, ready*} and all the output queues are empty. Performing the sequence *Udp_dreq2('1', '3 Alex Conf')* via *U_SAP2*, *Udp_dreq ('2', '3 Alex Conf')* via *U_SAP*, *Datareq('Hello')* via *C_SAP* leads to a global state in which an output is expected, respectively *Udp_dreq('2', '4 Hello')*. This output is a mapping of a *Data_Pdu* and it is contained in the queue U_SAP2_U. If this would be the only global state which could be reached the Timeout would not be justified. But there is another sequence which can be performed (see Figure 4.12) which is: *Udp_dreq 2 ('1', '3 Alex Conf')* via *U_SAP2*, *Datareq ('Hello')* via *C_SAP*, *Udp_dind ('2', '3 Alex Conf')* via *U_SAP*. This sequence differs from the first one by the fact that the internal transition *Udp_dind* is executed after the inputs *Answer_Pdu* and *Datareq* (not between them as in the first sequence). This sequence leads to a global state in which no output is expected (the output queues are empty). This state is quiescent (using *ioco* terminology). For this reason the Timeout is specified for the MSC of Figure 4.11. Therefore the Timeout generated for mutant 836 did not represent a discovery of an error. In a similar style as presented above we did the analysis of Timeouts for the rest of mutants.

In conclusion, Autolink was able to detect 17 mutants. By deriving more test cases it will be possible to increase its error detecting capability. The analysis of the Timeouts was done manually and it was not that simple. This experiment suggests that an automatization of such analysis can be beneficial for the tool because the human effort will be reduced and the tool will be able to detect more errors for the same amount of test runs.

4.4.2 TorX, TGV and Phact testing

For a detailed description of the test activity with TorX see [BFdV+99]. With TorX two separate test activities were performed by using the LOTOS and PROMELA specifications.

Using the LOTOS specification, TorX was repeatedly running against the 27 mutants, until either a depth of 500 steps was reached or an inconsistency between TorX and an implementation was detected (i.e., **fail**, usually after some 30 to 70 steps). On average, the time for a single step took 3 seconds on a 296 MHz Sun UltraSPARC-II processor. TorX was able to detect 25 mutants. The two mutants (444 and 666) that could not be detected accept PDUs from any source – they do not check whether an incoming PDU comes from a potential conference parter. This is not explicitly modeled in the LOTOS specification (and also in the SDL, PROMELA and FSM specifications), and therefore these mutants are **ioco**-correct with the correct implementation, which is why they can not be detected.

With the PROMELA based TorX the experiments that were done with the LOTOS based TorX were repeated. The same results were received, but in shorter time (on average about 1.1 step per second). The PROMELA based TorX was able to detect the same 25 of the 27 mutants as the LOTOS based TorX.

For a detailed description of the TGV test activity see [BRS+00]. We will outline the key elements of this experiment below. TGV used the LOTOS specification of the Conference Protocol for deriving tests. For defining test purposes two approaches were followed: 1) the test purpose was designed manually and 2) the test purposes were generated automatically.

For the first approach, 11 basic test purposes were generated for basic protocol functionalities (such as joining, leaving and data transfer) and 8 more complex test purposes. The time effort spent on designing and writing these 19 test purposes, generating the correspondent test suite with TGV and executing it against the mutants was 4 hours. Because one mutant went undetected, it was tried to be created 7 new test purposes. After 10 hours of work the new test design process was stopped.

For generating the test purposes automatically the following steps were performed: the CADP simulator was used to simulate the specification randomly. With this tool, 200 traces of 200 steps were produced and were translated into test purposes with a script. The test suite generated for these test purposes was able to detect all the *ioco* mutants. In [BRS+00] the timing for this approach is not reported.

For a detailed description of the testing activity performed with Phact see [HFT00]. We will sketch the key elements of this testing experiment. Using Phact, 82 tests were generated from the EFSM specification. The length of the test cases varied from 6 to 16 events. The timing for Phact is not reported in [HFT00]. The 82 test cases were successively applied to the set of mutants. When executing, 21 *ioco*-mutants were detected, and the rest of the mutant went undetected. The mutants 289, 293, 398, 444, 666 and 749 were the ones which were not detected.

4.5 Conclusions

In this chapter we have studied the feasibility of automatic test derivation of four tools: Autolink, TorX, TGV and Phact. To conduct this study, a protocol has been modelled in four formal specification languages: SDL, LOTOS, PROMELA and FSM. Also, a set of concrete implementations has been constructed, some of which were injected with faults that were unknown to the person that assisted in the test process. The results have been compared with respect to the number of erroneous implementations that could be detected for each of the tools, and the time and effort that it took to test the implementations.

We recall that in Chapter 3 we classified the four tools according to their conformance relations. Roughly speaking, it was concluded that TorX and TGV has the same detection power, Autolink has less detection power than the first two but it can be expected to have more detection power than Phact. These were the theoretical results. The results of the experiment are summarized below. It can be observed that a criterion of comparison is common for both the theory and the experiment, namely the number of errors which can be detected by a tool. In addition the experiment takes into account another criterion, namely the required computational effort for generating and executing the test cases.

First we will illustrate the main comparison results for Autolink and TorX. In the on-the-fly approach of TorX tests were fully automatically generated (in a random way) and executed. In the batch approach of Autolink the construction of tests needed manual assistance. Execution in the batch approach was done automatically. Both the on-the-fly approach and the batch approach were able to detect many erroneous implementations. Using the on-the-fly techniques all erroneous implementations could be detected, except for those that contained errors that simply could not be detected due to modelling choices in the specification and the choice of implementation relation. Using batch testing based on SDL fewer erroneous implementations were detected. On the one hand this is caused by the occurrence of Timeouts and on the other hand because fewer tests were executed due to the fact that manual assistance during test generation was needed. By deriving more test cases in the batch approach it will be possible to increase the error detecting capability. However, the results in this chapter support the assumption that if the test runs 'sufficiently long' then eventually all errors will be found. In the batch approach more human assistance is needed. Consequently, an error can often be found in less steps using the batch approach than in the on-the-fly approach.

When the test purposes were generated automatically, TGV was able to detect the same number of mutants as TorX. Phact was able to detect fewer mutants. While Autolink, TGV and TorX are able to generate larger test suites than the ones reported in this chapter, Phact reached the maximum size of its generated test suite. Therefore, Phact can not detect more mutants than the ones which it detected. There are also strong similarities between the manual approach for TGV and Autolink. In both cases, the time needed for the manual creation of test purposes was significant. The numbers of the test purposes which were created and transformed in test cases were small. For Autolink this depended also on the memory of the computers used. Consequently not all mutants could be detected (in both situations), some of them could have been found by larger test suites.

The experimental results that are presented in this chapter are based on a case study of a single protocol and a limited number of implementations. To obtain more valuable results the number of cases studies and the number of experiments per case study should be increased. To enable a rigid comparison of test generation and test execution tools by different vendors one (or more) case studies containing specifications and sets of implementations can be made publicly available, so that they can be used by several tool vendors to compare their test generation/execution tools with each other. The case study in this chapter can be seen as one of the first initiatives towards such a test tool benchmarking activity.

Chapter 5

Probabilities in the TorX test derivation algorithm

5.1 Introduction

One goal of the CdR project was to increase the performance of the prototype tool TorX developed within this project. One way for reaching this goal was by adding a suitable control mechanism to the test generation process of TorX. This can be done by extending TorX with explicit probabilities. Using these probabilities, the generated test suite can be tuned and optimized with respect to the probabilities of finding errors in the implementation. This chapter presents a probabilistic extension of TorX.

The TorX test generation tool is based on the *ioco* theory (see Chapter 2). In the heart of the theory is the *ioco* relation, which formally expresses the assumptions about stimulation and observation during testing. An algorithm for deriving a sound and complete test suite with respect to this relation forms the center of the TorX test generation tool. This algorithm is described in Section 2.2.

The algorithm is non-deterministic in the sense that in every state where the system can do both an input and an output, a choice must be made between these two. In practice a random generator was used to resolve this non-determinism, which resulted in an equal distribution of chances.

Practical experiments showed that in some cases this equal distribution served very well, but in other cases we encountered an anomalous situation. A case study, concerning an elevator, indicated that the derived test suite was not optimal. Analysis showed that the test suite mostly contained rather uniform test cases with respect to the ratio of inputs and outputs, thereby neglecting a collection of unbalanced behaviours which were very interesting for this particular case study. The natural solution to this problem is to extend the test derivation algorithm with explicit probabilities.

This research on the role of probabilities in test derivation is also inspired by our experiments, performed with the SDT tool set from Telelogic (see [SKGH97]), on testing the Conference Protocol. This case study also showed that a poor test suite may result when simply selecting at random between inputs and outputs.

These are the main motivations for the research presented in this chapter. We will study the impact of parameterizing the TorX test derivation algorithm with the probabilities of selecting between inputs and outputs. Furthermore, we will derive the optimal values for these probabilities given a desired ratio between inputs and outputs in the test cases.

This chapter is structured as follows. We refer to Chapter 2 for a description of the *ioco* theory and TorX. The proposed modification is presented in Section 5.2. Here we also calculate optimal values for the probabilities. In Section 5.3 we present a simple example. Our findings are summarized in

Section 5.4.

5.2 Adding probabilities

Our optimization of the TorX algorithm assigns probabilities p_1, p_2 and p_3 to the three choices of the algorithm. To get started, we assume that the probabilities p_1, p_2 and p_3 are global by which we mean that they do not depend on the specific moment of generation. Furthermore, we have:

$$p_1 + p_2 + p_3 = 1, p_1 \neq 0, p_2 \neq 0, p_3 \neq 0$$

The modified TorX algorithm now reads as follows:

- Take *Choice 1* (*terminate the test case*) with probability p_1;
- Take *Choice 2* (*supply an input for the implementation*) with probability p_2; select every input with the same probability; (we assume a finite input domain);
- Take *Choice 3* (*check the next output of the implementation *) with probability p_3.

An important observation is that the extended algorithm still produces the same test cases. We only control the chance of a trace to occur. This means that it keeps the properties of the old algorithm (Theorem 2.1.11): a generated test-case is finite, sound and the set of all tests is exhaustive.

After having extended the algorithm with probabilities, the question which arises is: what value should we give to these probabilities? The answer to this question is related to the ratio of inputs and outputs. Given a required ratio between the inputs and the outputs in a test trace what values should the probabilities of sending an input and receiving an output have? The answer will be formulated by the Lemma and the Theorem that follow.

Lemma 5.2.1 will provide a formula for the probability that the algorithm will arrive at the end of one given trace. Theorem 5.2.2 will compute a configuration for p_1, p_2 and p_3 which maximizes the probability to arrive at the end of one given trace.

Now, for a good understanding, we will define a special tree which will be used in the subsequent proofs. We call this tree *the behaviour tree* and it is formed by the union of all the traces derived from a specification S (extended with ϵ). An example of this tree is given in Figure 5.1.
The behaviour tree is composed of the following kinds of nodes:

- *Final:* in this node the trace of the test stops with a verdict (**pass**, **fail**);
- *Intermediate:* this node contains the name of the input or of the output or of the null output δ.

In the behaviour tree of Figure 5.1 the **pass** final state appears twice in one level because one **pass** verdict can be generated from *Choice 1* and one from *Choice 3*. When we refer to a *given trace* we refer to a trace of this tree which starts from the *Root* state and stops somewhere in the tree (the traces can be infinite). The signals from the trace are mapped to the non-root nodes of the behaviour tree. In the behaviour tree all the tests generated by the algorithm are included. Each intermediate node of the tree is reached through a trace which is going from the root to that intermediary node. After performing this trace, an implementation can produce an output (from a subsequent node of the intermediary node considered) with a given probability.

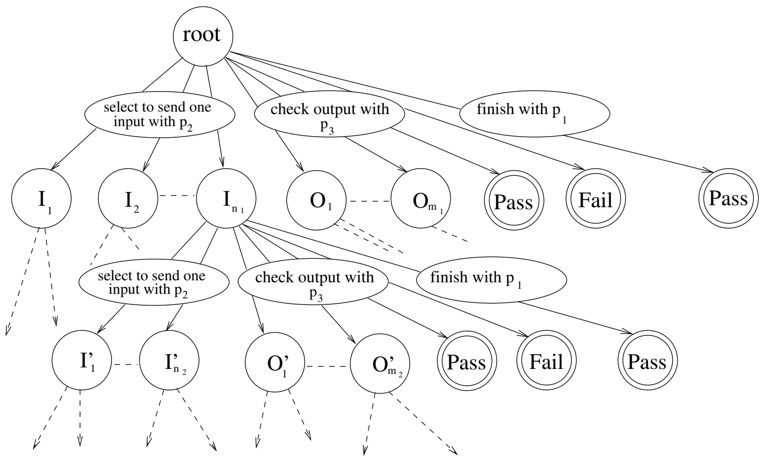


Figure 5.1: The behaviour tree for the extended TorX algorithm.

Lemma 5.2.1 *Consider an arbitrary but fixed finite trace which does not end in a final verdict. Let n be the number of inputs on the trace and p the length of the trace. Let r_l, $l = 1, 2, 3..n$ be the number of inputs which can be selected when the l-th input on the trace is selected. Let P_k, $k = n+1, 2, 3..p$ be the probability of the $(k-n)$-th output in the trace to be produced by the implementation. Then the probability P to generate this trace with the TorX algorithm is computed in the following way:*

$$P = \prod_{l=1}^{n}(\frac{1}{r_l} \times p_2) \times \prod_{k=n+1}^{p}(P_k \times p_3)$$

A good illustration is given in the example from Figure 5.2. A formal elaboration of the proof can be found in Appendix A.1.

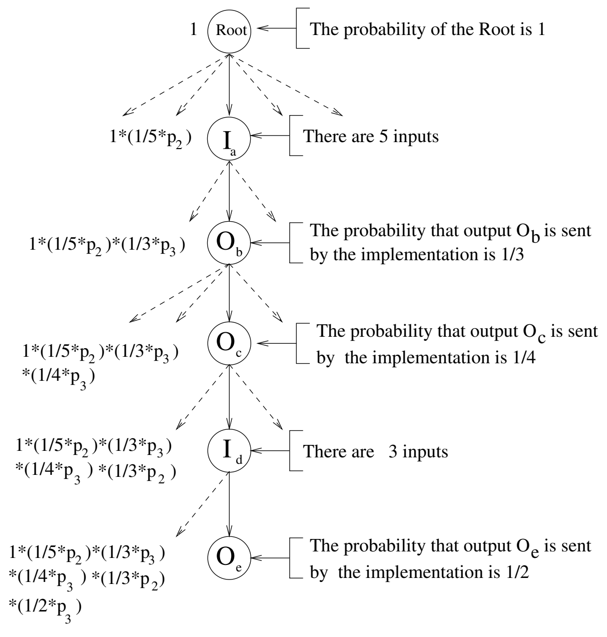

1
Root
The probability of the Root is 1
1*(1/5*p_2)
I_a
There are 5 inputs
1*(1/5*p_2)*(1/3*p_3)
O_b
The probability that output O_b is sent by the implementation is 1/3
1*(1/5*p_2)*(1/3*p_3) *(1/4*p_3)
O_c
The probability that output O_c is sent by the implementation is 1/4
1*(1/5*p_2)*(1/3*p_3) *(1/4*p_3) *(1/3*p_2)
I_d
There are 3 inputs
1*(1/5*p_2)*(1/3*p_3) *(1/4*p_3) *(1/3*p_2) *(1/2*p_3)
O_e
The probability that output O_e is sent by the implementation is 1/2

Figure 5.2: An example of computing the probability to generate a given trace.

Example In the considered trace there are two inputs I_a, I_d and three outputs O_b, O_c and O_e. The number of all inputs which can be selected when I_a is selected is five and for I_d it is three. Then the probability that the input I_a or the input I_d is chosen from the set of inputs is $\frac{1}{5}$ respectively $\frac{1}{3}$ (independent events). The probability that the implementation sends the output O_b, O_c or O_e is $\frac{1}{3}$, $\frac{1}{4}$ and $\frac{1}{2}$ respectively. The probability of arriving in the *Root* state is always one. In the computation by S_1 we mean the root node, S_2 stands for the node which contains I_a, S_3 for O_b, S_4 for O_c, S_5 for I_d and S_6 for O_e. With this the computation of arriving at the end of this trace is:
$P(S_1) = P(\text{Root}) = 1$
$P(S_2) = P(S_1) \times P(\text{Select input } I_a) \times P(\text{Choice } 2) = 1 \times (\frac{1}{5} \times p_2)$
$P(S_3) = P(S_2) \times P(O_b) \times P(\text{Choice } 3) = 1 \times (\frac{1}{5} \times p_2) \times (\frac{1}{3} \times p_3)$
$P(S_4) = P(S_3) \times P(O_c) \times P(\text{Choice } 3) = 1 \times (\frac{1}{5} \times p_2) \times (\frac{1}{3} \times p_3) \times (\frac{1}{4} \times p_3)$
$P(S_5) = P(S_4) \times P(\text{Select input } I_d) \times P(\text{Choice } 2) = 1 \times (\frac{1}{5} \times p_2) \times (\frac{1}{3} \times p_3) \times (\frac{1}{4} \times p_3) \times (\frac{1}{3} \times p_2)$
$P(S_6) = P(S_5) \times P(O_e) \times P(\text{Choice } 3) = 1 \times (\frac{1}{5} \times p_2) \times (\frac{1}{3} \times p_3) \times (\frac{1}{4} \times p_3) \times (\frac{1}{3} \times p_2) \times (\frac{1}{2} \times p_3) = \prod_{l=1}^{2} (\frac{1}{n_l} \times p_2) \times \prod_{k=3}^{5} (P_k \times p_3)$
with $n_1 = 5, n_2 = 3, P_3 = \frac{1}{3}, P_4 = \frac{1}{4}, P_5 = \frac{1}{2}$.

Once we have the formula for the probability of generating a trace we can look for the optimal configuration of the global probabilities in function of a given ratio between inputs and outputs. The following theorem solves this optimality problem.

Theorem 5.2.2 *Consider an arbitrary trace which does not end in a final verdict. Let n be the number of inputs on the trace and m the number of outputs on the trace* $(n, m \geq 1)$*.*

a) The probability to generate this trace reaches a maximum for $p_1 \to 0$, $p_2 = \frac{n}{n+m} \times (1 - p_1)$ *and* $p_3 = \frac{m}{n+m} \times (1 - p_1)$ *(we will explain about* $p_1 \to 0$ *at the end of this section);*

b) For every trace with ratio between inputs and outputs $r = \frac{n}{m}$ *the probability to generate this trace reaches a maximum for* $p_1 \to 0$, $p_2 = \frac{r}{r+1} \times (1 - p_1)$ *and* $p_3 = \frac{1}{r+1} \times (1 - p_1)$.

Proof:

a) The probability to generate a given trace is a function of two variables p_2 and p_3 (Lemma 5.2.1). For obtaining the extremal values for this probability the differential of the probability must be 0. Before doing this we will change the probability $P(p_2, p_3)$ to depend on p_1, p_2, writing $P(p_1, p_2)$. From

$$p_1 + p_2 + p_3 = 1$$

we derive

$$p_3 = 1 - p_1 - p_2$$

Conform Lemma 5.2.1:

$$P(p_2, p_3) = \prod_{l=1}^{n} (\frac{1}{r_l} \times p_2) \times \prod_{k=n+1}^{n+m} (P_k \times p_3) = p_2^n \times p_3^m \times \prod_{l=1}^{n} \frac{1}{r_l} \times \prod_{k=n+1}^{n+m} P_k$$

Therefore

$$P(p_1, p_2) = p_2^n \times (1 - p_1 - p_2)^m \times \prod_{l=1}^{n} \frac{1}{r_l} \times \prod_{k=n+1}^{p} P_k$$

We observe that $\prod_{l=1}^{n} \frac{1}{r_l}$ and $\prod_{k=n+1}^{n+m} P_k$ are constants which will be called C_1 and C_2. The differential of $P(p_1, p_2)$ is:

$$dP(p_1, p_2) = \frac{\partial P}{\partial p_1}(p_1, p_2)dp_1 + \frac{\partial P}{\partial p_2}(p_1, p_2)dp_2$$

We want both derivatives to be equal to 0.

$$\begin{cases} \frac{\partial P}{\partial p_1}(p_1, p_2) = 0 \\ \\ \frac{\partial P}{\partial p_2}(p_1, p_2) = 0 \end{cases}$$

Then

$$\frac{\partial P}{\partial p_1}(p_1, p_2) = C_1 \times C_2 \times m \times (-1) \times p_2^n \times (1 - p_1 - p_2)^{m-1}$$

and

$$\frac{\partial P}{\partial p_2}(p_1, p_2) = C_1 \times C_2 \times (n \times p_2^{n-1} \times (1 - p_1 - p_2)^m + m \times (-1) \times p_2^n \times (1 - p_1 - p_2)^{m-1})$$

Putting the first derivative equal to zero yields

$$C_1 \times C_2 \times m \times (-1) \times p_2^n \times (1 - p_1 - p_2)^{m-1} = 0$$

Then

$$\begin{cases} p_2 = 0 & \text{or} \\ p_2 = 1 - p_1 \end{cases}$$

The points are $(p_1, 0)$ and $(p_1, 1 - p_1)$. Putting the second derivative equal to zero yields

$$C_1 \times C_2 \times p_2^{n-1} \times (1 - p_1 - p_2)^{m-1} \times (n \times (1 - p_1 - p_2) - m \times p_2) = 0$$

Then

$$\begin{cases} p_2 = 0 & \text{or} \\ p_2 = 1 - p_1 & \text{or} \\ n - n \times p_1 - n \times p_2 - m \times p_2 = 0 \Rightarrow p_2 = \frac{n}{n+m} \times (1 - p_1) \end{cases}$$

The points are $(p_1, 0)$, $(p_1, 1 - p_1)$ and $(p_1, \frac{n}{n+m} \times (1 - p_1))$.
Now the point of maximum is for

$$P(p_1, \frac{n}{n+m} \times (1 - p_1)) = C_1 \times C_2 \times (\frac{n}{n+m})^n \times (1 - \frac{n}{n+m})^m \times (1 - p_1)^{n+m}$$

($P(p_1, 0) = 0$ and $P(p_1, 1 - p_1) = 0$ give points of minimum)
We have the following:

- for P maximal we need $(1-p_1)^{n+m}$ maximal; therefore $p_1 \to 0$;
- $p_3 = 1 - p_1 - p_2 = \frac{m}{n+m} \times (1-p_1)$.

b) Let us consider a finite trace with ratio $\frac{n}{m}$ where n is the number of inputs and m the number of outputs on the trace. Then to maximize the probability to generate this trace we have (point a)) $p_2 = \frac{n}{n+m} \times (1-p_1) \Rightarrow p_2 = \frac{\frac{n}{m}}{\frac{n}{m}+1} \times (1-p_1)$ and
$p_3 = \frac{m}{n+m} \times (1-p_1) \Rightarrow p_3 = \frac{1}{\frac{n}{m}+1} \times (1-p_1)$.
Now, because $r = \frac{n}{m}$, we obtain the formulas of point b) of the theorem: $p_2 = \frac{r}{r+1} \times (1-p_1)$ and $p_3 = \frac{1}{r+1} \times (1-p_1)$. $\square$

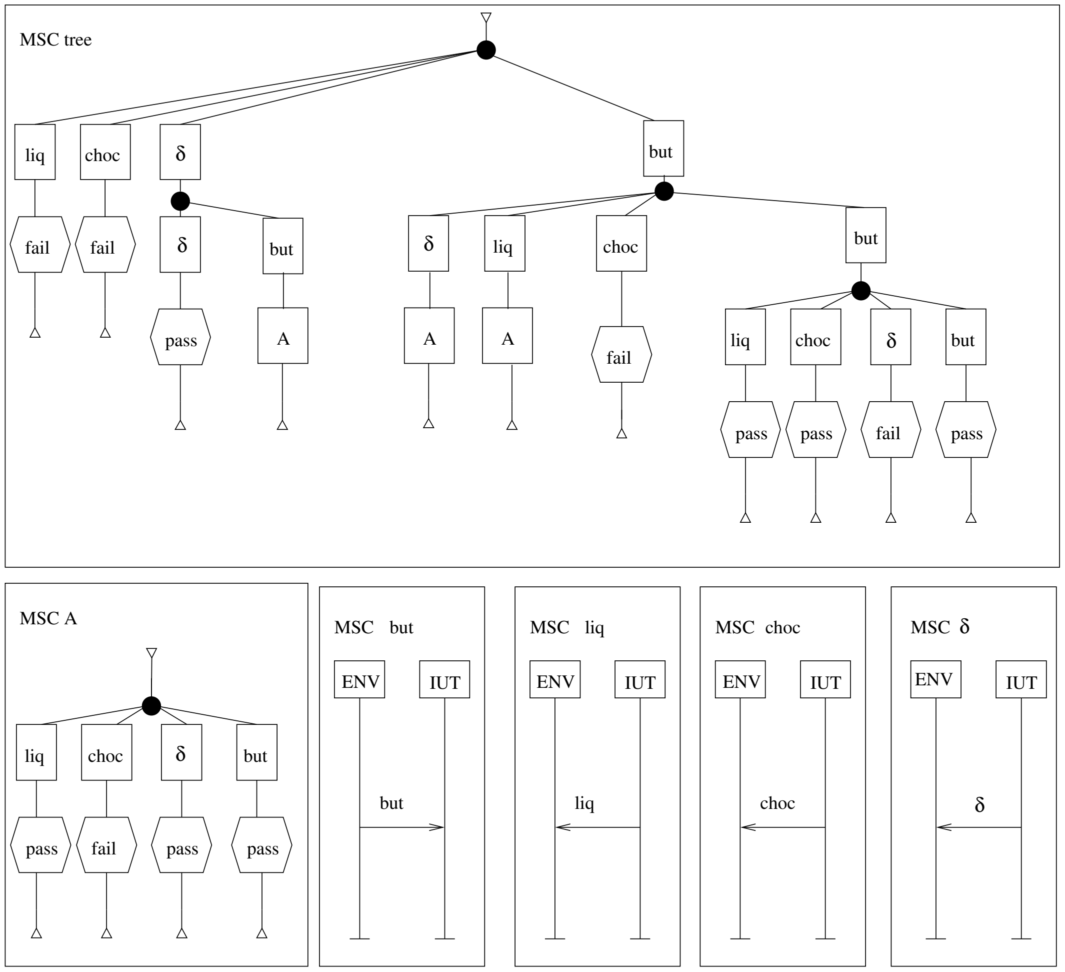


Figure 5.3: Tests derived from candy machine represented in an HMSC.

In the theorem we have that $p_1 \to 0$ for P maximal. We did not put $p_1 = 0$ in order to give to the TorX algorithm a possibility to finish. We remind that p_1 is global, which means that its value does not change in time. In practice, this can be turned into $p_1 = 0$ in different ways. For example, as TorX is doing in practice, two configurations for p_1, p_2 and p_3 can be used. In the first configuration the probabilities are $p_1 = 0$ and $p_2, p_3 \neq 0$ till a predefined length limit of the generated trace is reached. The results of the theorem can be applied because p_1, p_2, p_3 do not change values (in the generation period they are global). When the length limit is reached the generation and execution stops ($p_1 = 1$, $p_2 = p_3 = 0$; this is the second configuration).

5.3 Application

In this section we will work out an example for the probabilistic theory developed up to this point in this chapter. Let us consider all traces of the tests generated from the candy machine from Figure 2.1 with a length less than or equal to three. In practice, these traces can be obtained by using TorX in batch mode. They are the traces of the exhaustive test suite generated with TorX for the candy machine with a length less than or equal to three. These traces are represented in the HMSC (see [MR97]) from Figure 5.3.

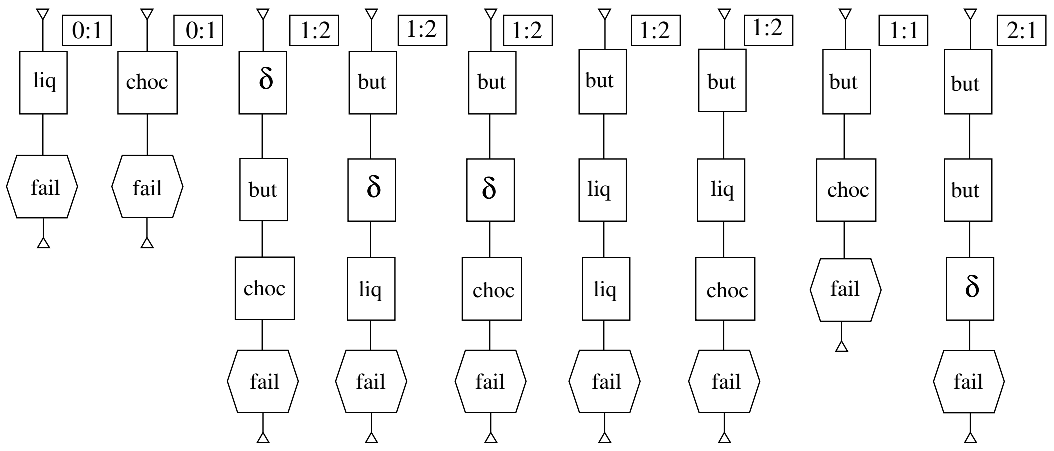


Figure 5.4: **Fail** traces represented in HMSC.

We use HMSC (High level Message Sequence Chart) to represent the test cases because this is a convenient technique which supports reusing parts of the diagram.

In the HMSC the **fail** traces {δ *liq*, δ *choc*, δ δ *liq*, δ δ *choc*} are not represented because only choosing to check the outputs will not lead to interesting test cases (so for the sake of simplicity we skipped some of them). Our example works even if these traces are present in the set of **fail** traces considered.

The set of all the **fail** traces are represented in Figure 5.4. In this figure, also the ratio between the number of inputs in that trace and the number of outputs is represented. For example the trace *but* δ *liq* has one input and two outputs. Therefore its ratio is $\frac{1}{2}$. The same procedure is applied to every trace in the set.

In this set of **fail** traces there are two traces with a ratio between inputs and outputs of $\frac{0}{1}$, five with a ratio $\frac{1}{2}$, one with ratio $\frac{1}{1}$ and one with ratio $\frac{2}{1}$. It is clear that the number of traces with ratio $\frac{1}{2}$ is the largest and we will choose it to be the ratio between inputs and outputs ($\frac{n}{m} = \frac{1}{2}$). For computing the new configuration of the probabilities (Theorem 5.2.2) we choose $p_1 = 0$ if the length of the trace is less than three and $p_1 = 1$ if the length is equal to three. Because the theorem applies to traces which do not end in final verdicts, in the computation of p_2 and p_3, p_1 will be zero. By applying Theorem 5.2.2 b) we obtain:

$p_2 = \frac{\frac{1}{2}}{\frac{1}{2}+1} \times (1 - 0) = \frac{1}{3} \approx 0.33$

and

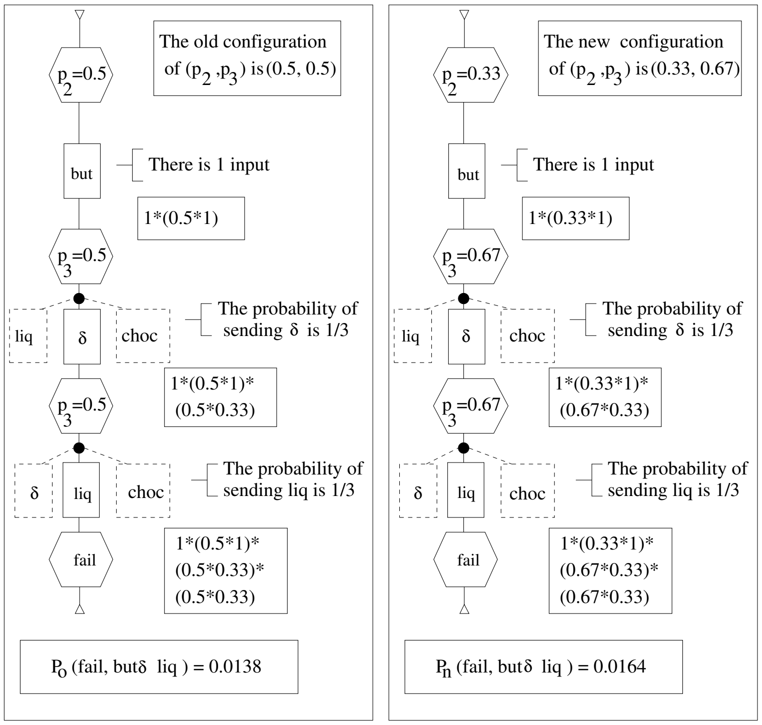

The old configuration of (p2 ,p3) is (0.5, 0.5)
p2=0.5
but
There is 1 input
1*(0.5*1)
p3=0.5
liq
δ
choc
The probability of sending δ is 1/3
1*(0.5*1)* (0.5*0.33)
p3=0.5
δ
liq
choc
The probability of sending liq is 1/3
fail
1*(0.5*1)* (0.5*0.33)* (0.5*0.33)
Po (fail, butδ liq) = 0.0138
The new configuration of (p2 ,p3) is (0.33, 0.67)
p2=0.33
There is 1 input
1*(0.33*1)
p3=0.67
The probability of sending δ is 1/3
1*(0.33*1)* (0.67*0.33)
p3=0.67
The probability of sending liq is 1/3
1*(0.33*1)* (0.67*0.33)* (0.67*0.33)
Pn (fail, butδ liq) = 0.0164

Figure 5.5: The probability of generating and executing the trace *but δ liq*.

$p_3 = \frac{1}{\frac{1}{2}+1} \times (1-0) = \frac{2}{3} \approx 0.67$

The old configuration of the TorX algorithm of (p_2, p_3) was (0.5, 0.5); the new one is (0.33, 0.67). For computing the probability of getting a **fail** when the algorithm runs one time against an erroneous implementation (which has all the **fail** traces from the set) first the probability of every individual **fail** trace should be computed. The probability that the TorX algorithm generates and executes a trace is given by Lemma 5.2.1. A graphical representation for the computation of the probability of the trace (*but δ liq*) is given in Figure 5.5 for the old and the new configuration of (p_2, p_3).

After performing the trace *but*, the IUT can send three outputs δ, *liq*, *choc*. Therefore the probability of sending one of them, such as δ, is 0.33. In the same way the probability of sending *liq* is also 0.33. By applying the lemma it results that the probability of generating and executing the trace *but*

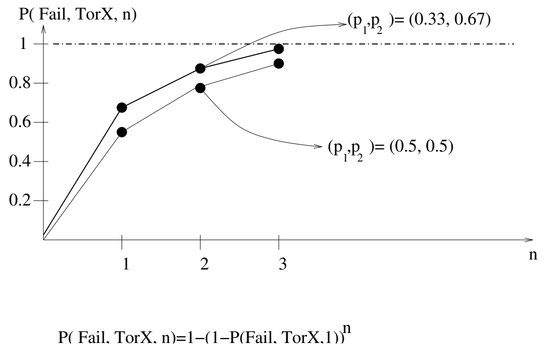


Figure 5.6: The probability of getting a **fail** in *n* test generation-executions.

$\delta\ liq$ is 0.0138 for the old configuration of the probabilities and 0.0164 for the new one. In a similar way the probabilities for every individual trace which ends in a **fail** are computed.

It is not entirely trivial to see that optimizing the chance of generating each individual **fail** trace leads to a better error detection capability for the suite as a whole. In order to show that this is the case, we made some further calculations in the context of this example.

The probability P(**fail**, TorX, 1) of getting a **fail** verdict when the TorX algorithm runs once against the IUT is obtained by summing the probabilities of every individual **fail** trace; so for the old configuration this probability is P_{old}(**fail**, TorX, 1) $= 0.51$ and for the new configuration it is P_{new}(**fail**, TorX, 1) $= 0.62$. This simple case clearly demonstrates that a modification of the probabilities can lead to a higher chance of discovering an erroneous implementation in the same number of algorithm runs. This is also clear from the graph in Figure 5.6 in which the probability of getting a **fail**, denoted as P(**fail**, TorX, n), as a function of the number n of test generation-executions is expressed (for the old and for the new probabilities configuration).

5.4 Conclusions

In this chapter we proposed to modify the TorX test derivation algorithm such that the probabilities of the non-deterministic alternatives are made explicit.

We argued that in some cases the generated test suite can be optimized by adapting the values of these probabilities. Case studies gave evidence that assuming an equal distribution of chances, the TorX algorithm will sometimes yield relatively few really interesting test cases.

An important question is, of course, whether there are heuristics which help in selecting appropriate values for the probabilities. In the case studies which we performed, the ratio between the number of inputs and the number of outputs in a test trace influenced the quality of the test cases. Therefore, we derived in this chapter the optimal values for the probabilities in the algorithm given some preferred ratio between the number of inputs and outputs. Our calculations on the toy example of the candy machine confirm that an appropriate choice of the probabilities improves the ability to detect errors in the implementation.

The proposed modification of the TorX algorithm has already been implemented. In the next chapter we will study the experiment with the Probabilistic TorX on the Conference Protocol case study.

Chapter 6

Experimenting with the probabilistic TorX algorithm

6.1 Introduction

In Chapter 5 (see also [FGM00]) we presented a generalization of the TorX test derivation algorithm with probabilities. The natural continuation of this theoretical work is to investigate the results experimentally. This chapter presents our findings when experimenting with the probabilistic TorX on the Conference Protocol case study, findings which were also presented in [Gog03a].

In Chapter 5, we compared the performances of TorX and of the probabilistic TorX on the toy example of the candy machine. In this example we looked at the fail-traces of the candy machine which are generated by TorX and which do not have a length greater than 3. We found that they have similar ratios and we choose the ratio which occurred the most. Using this ratio we computed the probabilities p_1, p_2 and p_3 for the three choices of TorX. Going forward with the probabilistic computation, we found out that the probabilistic TorX has a greater probability to generate a **fail** than the original TorX.

The comparison in Chapter 5 is based on a theoretical example. We wanted also to experiment with the probabilistic TorX on a real case study. For experimenting we had to choose some parameters with which TorX works. So we had to choose the specification and the set of implementations which are used by TorX when it derives tests on-the-fly. The first year of CdR was dedicated to the experiment with the Conference Protocol case study ([BFdV$^+$99], also Chapter 4); therefore to extend that experiment with the probabilistic TorX was quite natural.

In the experiment with the Conference Protocol, we wanted to investigate whether adding probabilities to TorX makes it produce better results (the *claim* of the [FGM00] theory), for example in the number of errors detected and doing that with shorter test cases. The experiments support the claimed benefits of probabilistic TorX over TorX if in the probabilistic setting more errors are found then in the normal setting.

The Conference Protocol specifications are described in Section 4.2. The original TorX and the probabilistic TorX are presented in Chapters 2 and 5. In this experiment we used two sets of mutants. The first one was used in the benchmarking experiment described in Chapter 4. The second one was made after this experiment and it is presented in Section 6.2. Section 6.3 presents the experiment itself and the comparison results. The conclusions are drawn in Section 6.4.

6.2 Conference Protocol implementations

In the experiment with the probabilistic TorX on the Conference Protocol, we had two sets of mutants. The division of the implementations in two sets corresponds to the temporal order in which the implementations were created. The two sets induced two complete separate phases in this experiment. First we experimented with the first set and after that with the second set. We will not merge these sets into one big set and we will present the two phases of the experiment separately according to the temporal order in which they were performed. The reason of separating them will become clear at the end of the next section.

The first set, which was also used in the experiment described in [BFdV$^+$99], consists of 28 different conference protocol implementations. One of these implementations is correct whereas 27 of them are erroneous. The first set of mutants is divided in three groups: *No outputs*, *No internal checks* and *No internal updates*. In Section 4.2.4 the main characteristics of these three groups are explained.

The second set of mutants is larger than the first set and consists of 43 mutants, all erroneous. These mutants can be divided in 6 groups. The first three groups are: *No outputs* (6 mutants), *No internal checks* (9 mutants) and *No internal updates* (4 mutants). In the second set, three more groups *No inputs* (12 mutants), *Wrong outputs* (9 mutants) and *Modifications on PDU information* (3 mutants) are present. The group *No inputs* is formed by mutants which do not perform the actions associated with specific inputs correctly. The group *Wrong outputs* is composed by mutants which send non-specified outputs after performing specific inputs. The group *Modifications on PDU information* is formed by mutants for which the information of the PDU is changed.

6.3 The experiment

In our experiment we had a LOTOS specification of the Conference Protocol and two sets of mutants. The first set was formed by 27 erroneous mutants and the second one by 43 erroneous mutants. Because the mutants and the specification were built by the UT partners in CdR, we did not know at that point of time how they were made and what specific errors they contained. The experiment was done in two phases: first we ran TorX and the probabilistic TorX against the first set of mutants and after that we did the same for the second set of mutants. Therefore, we will present the comparison results of TorX and the probabilistic TorX first for the first set of mutants and then for the second set.

When TorX is run against an erroneous implementation and an error is discovered, a **fail** verdict is generated. For our experiment we kept the configuration parameters which were used in [BFdV$^+$99]. Therefore the maximum number of steps, which means signals (inputs or outputs) of an execution trace, is 500. If the execution trace reached this limit the verdict was automatically **pass**. As soon as a **fail** was generated the test was aborted (in which case the length of the trace is less than 500).

Now, to investigate the claim of the theory, means to investigate whether the length of the fail-traces generated with TorX is on average greater than the length of the fail-traces generated with the probabilistic TorX. This can also have another consequence: when for some mutants the limit 500 is passed, TorX assigns a **pass**. If for these mutants the probabilistic TorX finds an error with a fail-trace which has a length below 500, the number of mutants detected with the probabilistic TorX will be greater than with TorX.

The probabilities are determined as fellows. Because TorX and the probabilistic TorX have a limit for the number of steps after which they automatically finish, the probability of Choice 1 is $p_1 = 0$ and $p_2 + p_3 = 1$, till 500. TorX has the same probability to select an input (Choice 2) as to check the output (Choice 3). This means that the probabilistic configuration for TorX is $p_2 = p_3 = \frac{1}{2}$. For the probabilistic TorX, when varying p_2 we obtain $p_3 = 1 - p_2$. For investigating the claim of the theory, we should vary p_2 and observe whether there are values of p_2 and p_3 for which the lengths of the fail-traces are generally lower than for the configuration $(p_2 = \frac{1}{2}, p_3 = \frac{1}{2})$ of TorX. For estimating the values of p_2, which give the points of minimum, we need to remember that Theorem 5.2.2 gives a relationship between the probabilities of TorX's choices and the ratio between inputs and outputs which occurs in an execution trace. Looking at the traces of the specification, we can try to guess good values for the ratio of the execution traces. After that, using Theorem 5.2.2, we can estimate the value of p_2 for which the length of the fail-traces has a minimum.

The comparison results for the first set of mutants We varied the probability p_2 using different values in the range from 0 to 1. We ran both the probabilistic TorX and TorX against every mutant once. For each mutant the same seed was used for both the probabilistic TorX and TorX. The seed was chosen randomly. This policy related to the seeds was maintained in all the experiments described in this chapter. The results with the probabilistic TorX and TorX are given in the *Table 1.* In this table, TorX is represented by its probabilistic configuration $(p_2 = \frac{1}{2}, p_3 = \frac{1}{2})$. In the table we did not put p_3 because it is easily computed as $1 - p_2$.

We also give a guess where the point of minimum average trace length should be. This is worked out for 20 traces of the specification which go from the initial state and finish in the initial state (a longer trace is composed of such traces). The traces were in part generated by the TAU/Telelogic tool when verifying the SDL Conference Protocol specification and some were representative use cases of the Conference Protocol chosen by the author. The analysis of the 20 traces showed that they have almost all ratios in the range $[1, 2]$ with an average ratio of 1.51. We assumed that the minimum should have the ratio in this range $[1, 2]$. We chose to do the experiment with the probabilistic TorX for the extremes of this range, respectively for the ratios 1 and 2. Looking at the ratios average and to some trace generated with the old TorX, which contained too many null outputs, we guessed that around 2, better results would be obtained. Our observation was that some traces generated with TorX contained sequences of repetitive null outputs. We remind that a null output is mapped into a timeout. To continue checking the output when only the null outputs is produced does not make too much sense (this will only multiply the timeout time). If the timeout was set correctly, it is not very likely that doubling or tripling it will make an implementation to produce an unexpected output. It is more likely that at the first timeout, an erroneous implementation will send an unspecified output. Therefore, in the case of a timeout, checking the outputs only once will be enough. Now, avoiding repetitive null outputs can be achieved by increasing the ratio such that more inputs will be present within a trace. Consequently, p_2 is larger. In this way, after a first null output it will be more likely that an input will be sent instead of checking another null output. Both analyses (of the 20 traces and of the sequences of null outputs) indicated that the ratio should be increased. The increase provided by the average of the 20 traces, viz. 1.51, is not very large. There is still a good chance that after a first null output another one will be checked again (using Theorem 5.2.2, the probability of checking the outputs, p_3, for the ratio 1.51 is 0.40). The extreme of the range, 2, provides a larger increase and for this reason we guessed that around this value better results will be obtained. The probabilities p_2 and p_3 for the ratio 2 are computed using Theorem 5.2.2: $p_2 = \frac{2}{2+1}(1 - 0) = 0.67$ and $p_3 = \frac{1}{2+1}(1 - 0) = 0.33$.

Table 1

Mutant	$p_2 = 0.33$ *verdict*	*length*	$p_2 = 0.5$ *verdict*	*length*	$p_2 = 0.58$ *verdict*	*length*	$p_2 = 0.67$ *verdict*	*length*	$p_2 = 0.75$ *verdict*	*length*
100	**fail**	36	**fail**	23	**fail**	21	**fail**	17	**fail**	146
111	**fail**	56	**fail**	34	**fail**	32	**fail**	23	**fail**	25
214	**fail**	162	**fail**	110	**fail**	91	**fail**	83	**fail**	75
247	**fail**	56	**fail**	34	**fail**	32	**fail**	23	**fail**	25
276	**fail**	36	**fail**	23	**fail**	21	**fail**	17	**fail**	25
289	**fail**	162	**fail**	110	**fail**	91	**fail**	86	**fail**	75
293	**fail**	108	**fail**	74	**fail**	65	**fail**	65	**fail**	480
294	**pass**	500	**pass**	500	**pass**	500	**pass**	500	**pass**	500
332	**pass**	500	**pass**	500	**fail**	486	**fail**	415	**pass**	500
345	**pass**	500	**pass**	500	**fail**	486	**fail**	409	**pass**	500
348	**fail**	496	**fail**	342	**fail**	153	**fail**	150	**fail**	238
358	**pass**	500	**pass**	500	**fail**	482	**fail**	411	**pass**	500
384	**fail**	108	**fail**	74	**fail**	65	**fail**	65	**fail**	44
398	**fail**	180	**fail**	115	**fail**	95	**fail**	91	**fail**	91
444	**pass**	500	**pass**	500	**pass**	500	**pass**	500	**pass**	500
462	**fail**	189	**fail**	120	**fail**	91	**fail**	114	**fail**	130
467	**fail**	106	**fail**	74	**fail**	65	**fail**	65	**fail**	44
548	**fail**	160	**fail**	104	**fail**	88	**fail**	84	**fail**	167
666	**pass**	500	**pass**	500	**pass**	500	**pass**	500	**pass**	500
674	**fail**	32	**fail**	20	**fail**	15	**fail**	12	**fail**	16
687	**pass**	500	**pass**	500	**pass**	500	**fail**	402	**pass**	500
738	**fail**	161	**fail**	116	**fail**	91	**fail**	114	**fail**	130
749	**fail**	36	**fail**	23	**fail**	21	**fail**	17	**fail**	17
777	**fail**	162	**fail**	110	**fail**	91	**fail**	83	**fail**	75
836	**fail**	36	**fail**	23	**fail**	21	**fail**	17	**fail**	146
856	**fail**	108	**fail**	74	**fail**	65	**fail**	17	**fail**	44
945	**pass**	500	**pass**	500	**fail**	483	**fail**	411	**pass**	500
Detected	19		19		23		24		19	

One can easily observe from *Table 1* that there are similarities in the way the errors in the mutants are discovered (for example, similar lengths for the first and the fifth mutant). We had a discussion regarding this phenomenon with the owners of this set of mutants from Twente University. It turned out that the same happened in the experiment reported in [BFdV$^+$99] and this is explained by the fact that some mutants have a very symmetric and similar structure. Also we found that two of these mutants can not be detected by TorX and the probabilistic TorX because they are non-*ioco* mutants, in other words they contain errors which can not be detected with the *ioco* relation on which TorX and the probabilistic TorX are based. The correct implementation is not included in the table (it was tested, of course).

The results show that most of the implementations have the minimum for $p_2 = 0.67$ which corresponds to the ratio 2 (as we guessed). The same thing is represented in Figure 6.1 where the graph *Lenght*(p_2) is shown for the mutant 100. By *Length* we mean the length of the generated trace. Similar graphics can be drawn for the rest of the mutants.

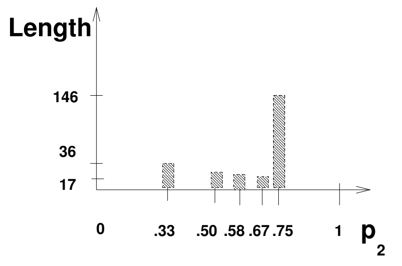


Figure 6.1: The graph of the length for one fail trace (mutant nr. 100).

Using the values of p_2 from *Table 1* we made some statistical computations. The statistical estimator mean $\overline{p_2}$ [All78], which is the average of the values of p_2 for which minima of the length are obtained is 0.678. Let us assume that the distribution of the minima for p_2 is a normal distribution (see Figure 6.2). In the figure, the number 20 is the number of mutants which have the minimum of the length for $p_2 = 0.67$, 5 corresponds to $p_2 = 0.75$ and 2 to $p_2 = 0.58$. The normal distribution is centered on the mean, 0.678. From the statistical computation it can be observed that the mean is only slightly different from the guessed value of 0.67. The standard deviation σ which is computed as $\frac{1}{26} \times \sum_{k \in \{100,\ldots,945\}} (p_2(k) - \overline{p_2})^2$ is 0.04. In the formula of σ, $p_2(k)$ is the value of p_2 for the mutant k ($k \in \{100, ..., 945\}$) for which the minimum of the length is obtained. The variance of $\overline{p_2}$ which is computed as $\frac{2\sigma}{\sqrt{n}}$ with $n = 27$ is 0.015. The variance is a measure of the tightness of the clustering about $\overline{p_2}$. It is expected that three-fourth of the values of p_2 (for which the length is minimum) are within the variance of $\overline{p_2}$ [All78], which is not further than 0.015 from the mean. In our case the 20 values of 0.67, which represent three-fourth of the total number of 27 mutants are within the expected range of $[0.678 - 0.015, 0.678 + 0.015]$.

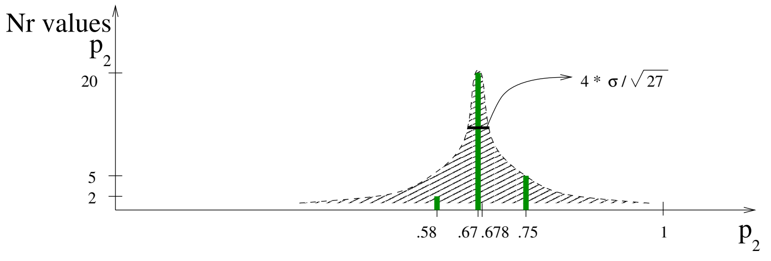


Figure 6.2: The distribution for p_2.

The same minimum is obtained when running TorX more times against each mutant. For example, in Figure 6.3 we represented the graph *Lenght*(p_2) obtained when the probabilistic TorX and TorX were run 20 times against the mutant 100. We represented in the figure the averages of the series of 20 lengths obtained for each p_2. As one can see the minimum of the *Length* occurs also for $p_2 = 0.67$.

The same conclusion can be obtained when looking at the number of mutants detected which is represented in Figure 6.4. For TorX we have 19 mutants detected (an error in every mutant) for one run of TorX against every mutant and for the probabilistic TorX using $p_2 = 0.67$ we have 24 mutants detected. TorX and the probabilistic TorX with $p_2 \in \{0.58, 0.66, 0.75\}$ were able to discover all the 25 *ioco* detectable mutants, after two runs. The probabilistic TorX with $p_2 = 0.33$ discovered after two runs 24 mutants and after three runs all the *ioco* mutants. Looking at what happened, we can conclude that the probabilistic TorX using $p_2 = 0.67$ detected almost all the *ioco* mutants from the first run. The claim of the theory is consistent with the experimental results because the probabilistic TorX using $p_2 = 0.67$ gives better results than TorX.

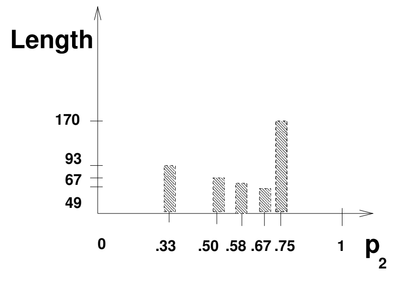


Figure 6.3: The graph of the average length for 20 runs (mutant nr. 100).

Exploiting the hypothesis The experiment which we did gives rise to the following idea on how to find the values of p_2 for which we can obtain better results. One can pick up a few mutants randomly from the set of mutants and make the curve $Length = f(p_2)$ for them. The values of p_2 for which there are points of minimum can be observed. Based on them, the mean can be computed. Thereafter, the mean can be used for the rest of the mutants. Let us assume that the mean is convergent, which means that for a sufficiently large number n of experiments the computed value of the mean is a good approximation of the limit. Usually for a size of 30, the computed value of the mean is a good estimation for the limit ([All78]). Therefore a recommended value for the number of mutants is 30. For this experiment in which we had in total 70 mutants, the number of implementations of 27 is a little bit lower than 30. We can use the computed value of the mean for the second set of mutants, but the approximation will be larger than when using 30 mutants. Because the value of the mean is 0.678 only slightly different from the guessed one of 0.67, we kept the guessed one for the second set of mutants.

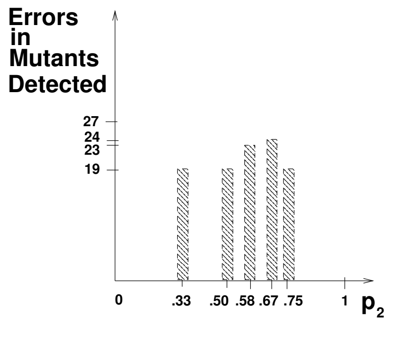


Figure 6.4: The graph of the number of mutants detected.

The comparison results for the second set of mutants We drew the same conclusion as from the first set of mutants when running TorX and the probabilistic TorX against the 43 mutants, all *ioco* detectable: almost all the mutants have the minimum of the *Length* for $p_2 = 0.67$. The mean in this case was 0.66 and the variance 0.02. The probabilistic TorX with $p_2 = 0.67$ discovered 42 mutants when we ran it once against every mutant, and TorX discovered 40 mutants. The experiment with the second set can be seen as a confirmation of our ideas expressed in the paragraph **Exploiting the hypothesis**. So the first set can be seen as an experimental set for finding the values of p_2 for which one can expect better results with the probabilistic TorX. The second set, larger than the first one, confirms that the probabilistic TorX gives better results using the mean of these values of p_2.

There is another important aspect to be mentioned regarding the second set of mutants. These mutants are discovered with fail-traces the lengths of which are usually shorter than the ones of the first set. For example, the lengths of the fail-traces of 35 mutants are less than 75 for the probabilistic TorX with $p_2 = 0.67$; for TorX this holds for 30 mutants. Increasing the length limit, the lengths of the fail-traces of 40 mutants are less than 170 for the probabilistic TorX with $p_2 = 0.67$; for TorX the number is 38. As can be seen, the gain for the second set was in getting shorter fail-traces. The difference in the number of mutants detected was not as big as for the first set. Fixing, for example, the length limit at 75 would increase the difference to 5 mutants. As can be easily observed, the reason is that the length limit was fixed to a value too high to make the difference more visible. To put it in practical terms: the potential advantage is in reducing the size of the generated tests rather than in finding more errors.

6.4 Conclusions

In this chapter we presented the experiment with the probabilistic TorX and the original TorX on the Conference Protocol. The experiment made with the Conference Protocol case study confirms that the performance of the TorX algorithm increases when probabilities are used. When comparing the length of the fail-traces which led to error detections, the probabilistic TorX when using $p_2 = 0.67$ produced shorter fail-traces than the original TorX. For the same value of p_2, the number of errors in mutants detected with the probabilistic TorX was greater than the number of errors in mutants detected with TorX. The value $p_2 = 0.67$ for which the probabilistic TorX produced better results corresponds to our guess made about p_2 when looking to the ratios between inputs and outputs of representative use cases of the Conference Protocol.

The experiment gives rise to another idea on how to find the values of p_2 for which better results can be obtained. One can choose few mutants and make the curve $Length(p_2)$ for them. Then observing the values of p_2 which give points of minimum, computing the statistical mean (the average of these values) and apply the value of the mean for the rest of the mutants.

The experiment itself confirms that this idea is valid. We tested two sets of mutants. The first set indicated the values of p_2 which gave points of minimum. After that we computed the mean. Applying the mean on the second set of mutants, larger than the first one, we obtained points of minimum closed to the value of the mean. But we should say that more experiments are needed for a more complete validation of this idea.

Chapter 7

A probabilistic coverage for on-the-fly test generation algorithms

Systematic testing is an important technique to check and control the quality of software systems. Testing consists of systematically developing a set of experiments or test cases and then running these experiments on the software system that has to be tested, i.e. the IUT. The subsequent observations made during execution are used to determine whether the IUT behaved as expected leading to a verdict about the IUT's correctness.

As described in Chapter 1, in a traditional approach the test generation and test execution are separated phases of the test experiment. We refer to such an approach as *batch-oriented*. A newer technique for test derivation is to combine the test generation and the test execution in one phase. We refer to such a technique as *on-the-fly*. An example of a tool which works on-the-fly is TorX [BFdV$^+$99].

An on-the-fly algorithm derives and runs a finite number of tests which can detect a certain amount of errors, with others going undetected. In order to express this, as ITU-T Z.500 recommends, the concept of *coverage measure* should be used (see Chapter 1). An example of such a coverage measure which is useful for a set of test cases can be found in [HT96].

The coverage measure from [HT96] has a probabilistic nature and it can be a applied for a batch-oriented test suite. We will explain below the stochastic nature of this coverage. Due to the non-determinism of the IUT, each run of a test may lead to a different outcome and the outcomes of several, independent test runs make up one observation. So, some outcomes are more likely to occur than others. The probability of an outcome to occur can be thought of as depending on the frequency with which the implementation resolves the nondeterministic choices leading to the different outcomes. This leads to the consideration of the occurrence of an outcome of a test run as a stochastic experiment. Also, the design and the implementation process of a complex system are viewed as a stochastic experiment. For example when designing a juice machine which is supposed to deliver juice and tea it is less likely to end up with a completely different kind of machine (as for example a washing machine) than to end up with a slightly different, but possibly incorrect, juice machine. Moreover, assuming that an implementer can make several independent mistakes with non-zero probability and while trying to do his task as good as possible, it is less likely for the implementer to make all possible mistakes than to make only a small number of mistakes. These are some examples which show that not every implementation has the same chance to occur as the result of 'implementing' a given specification. Also, to express the severity of the bugs in implementations, a weight assignment is added to the implementations. These three ingredients: the probability distribution of the outcomes produced by the test runs, the probability distribution of the implementations and the severity of the bugs in implementations form the base of the probabilistic coverage from [HT96].

The existing theory was useful for batch-oriented testing. Modern test process turns away from batch-oriented to on-the-fly. Therefore we generalized the existing theory from [HT96] for on-the-fly algorithms. The findings reported in this chapter were also presented at [Gog03b]. We adopted the same views as [HT96] regarding the probabilistic distribution of IUTs and the weights assigned to implementations. Regarding the probability distribution of the outcomes, first we remind that the test generation and test running are integrated in one phase by an on-the-fly algorithm. Therefore, the probabilistic distribution for outcomes has two independent sources, in this case. First, the non-determinism of the IUT gives a probabilistic nature to outcomes. Secondly, the test generation algorithm itself can have a probabilistic component (as it is the case for TorX). The probabilistic nature of TorX comes from its three non-deterministic choices, each choice could be chosen with a given probability at a given moment in the generation process. The probabilistic nature of the algorithm itself is another reason to add probabilities to outcomes. In Chapter 5, we showed how to compute the probabilities of the outcomes for TorX (the outcomes are traces in this case). In a similar way, the probabilities of the outcomes can be computed for other test generation algorithms. With these being said, our coverage for an on-the-fly algorithm is defined as follows. The severity of the bugs in implementations, the probability distribution of the IUT and the probability distribution of the test outcomes which is seen as a result of an integrated test generation and running process are combined in a probabilistic coverage formula for an on-the-fly algorithm. The coverage is parameterized with the number of tests derived. The abstract coverage formula is instantiated for the *ioco* theory of test derivation using results from Chapter 5 ([FGM00]).

The chapter is organized as follows. Section 7.1 introduces a framework for the definition of a coverage measure and the assumptions on which the theory presented is based. Section 7.2 formally defines a weight assignment to each implementation and the probability of an implementation to occur. In Section 7.3 the coverage formula is defined and Section 7.4 instantiates the probabilistic coverage for the *ioco* theory. The conclusions are presented in Section 7.5.

7.1 Automated testing

In this section we will present in more detail some basic notions regarding conformance, testing and test generation which were also described in Chapter 1. We will give also the assumptions on which we base the coverage measure theory for on-the-fly test generation algorithms.

Conformance The conformance of an implementation under test (IUT) with respect to a specification implies the assumption of a universe of implementations IMPS and a universe of formal specifications SPECS. Implementations are concrete, informal objects, such as pieces of hardware, or pieces of software. In order to be able to reason in a formal way about them, it is assumed that each implementation IUT $\in$ IMPS can be modelled by a formal object i_{IUT} in a formalism MODS, which is referred to as the universe of models. Conformance is expressed by means of an implementation relation **imp** $\subseteq$ MODS $\times$ SPECS. We use $I_s =_{\text{def}} \{i \in \text{MODS} \mid i \ \mathbf{imp} \ s\}$ for the set of **imp**-correct implementations of s, and $\overline{I_s} =_{\text{def}} \text{MODS} \setminus I_s$ for its complement.

Testing The test cases, sometimes denoted simply as tests, are formally specified as elements of a universe of test cases TESTS. A set of test cases is called a test suite. The process of running a test repeatedly against an implementation is called test execution. Each test run leads to an outcome. Test execution leads to an observation, which is the set of all possible outcomes produced by the test runs, in a domain of observations OBS. Let $\mathcal{O}$ be the set of possible test-run outcomes, then an observation is a set of outcomes: OBS $= \mathcal{P}(\mathcal{O})$. To each outcome a verdict is assigned by a verdict assignment function $v_t : \mathcal{O} \rightarrow \{\textbf{pass}, \textbf{fail}\}$, where t is a test. The verdict of an observation will be **pass** if all outcome-verdicts are **pass**. Test execution is modelled by a function $exec$: TESTS $\times$ MODS $\rightarrow$ OBS.

On-the-fly test generation An on-the-fly algorithm combines two phases, the test generation and the test running, into one common phase. We model this mathematically by considering the generation of a test and the run of it as being formalized by the function *genexec*. The test execution is already modelled by the *exec* function, so that we should proceed with formalizing the test generation. Let ALGS be the universe of all the algorithms that generate and run tests on-the-fly. The process of generating and running tests by an on-the-fly algorithm will be called on-the-fly test generation and execution. An on-the-fly algorithm $A \in$ ALGS is applied to a specification $s \in$ SPECS and to an implementation $i \in$ MODS. The set of all the tests, which we will denote as $A(s, i)$, derived with A when it uses the specification s and the implementation i is a subset of TESTS, i.e. $A(s, i) \subseteq$ TESTS. Now the observation made by an on-the-fly algorithm A is the set of all outcomes which can be produced by the runs of all the tests which can be generated with A, which is $\cup_{t \in A(s,i)} exec(t, i)$. Then let *genexec*: ALGS $\times$ SPECS $\times$ MODS $\rightarrow$ OBS be the function that correctly models the test generation and execution of an algorithm on-the-fly, then $genexec(A, s, i) = \cup_{t \in A(s,i)} exec(t, i)$.

The assumptions for on-the-fly test-generation Now we will introduce the assumptions which are at the base of all the coverage computations. We have the same probabilistic approach as in [BTV91] and [HT96], but our contribution is in extending it by considering the test generation and execution as a stochastic process, and in instantiating it for the *ioco* theory and for the specification of the TorX test-generation-algorithm. We focus on on-the-fly testing.

- $\mathcal{A}$: we assume that the occurrence of an implementation i is the outcome of a stochastic experiment; we adopt the notation $p_s(i)$ from [HT96] to denote the probability that implementation i occurs;

- $\mathcal{B}$: we assume that the generation of a test and the running of the test against an implementation i by an on-the-fly algorithm A, which uses a specification s, is a stochastic experiment; in particular $p_{A,s,i}(\sigma)$ is the probability that A generates a test which leads to outcome σ when it runs the test against i and uses a specification s.

7.2 Valuation of the implementation

Weight of implementations To express the importance of each implementation, a weight assignment ([HT96]) is added to each implementation. Let $s \in$ SPECS be a specification, and $\textbf{imp} \subseteq$ MODS $\times$ SPECS an implementation relation, then a function w : MODS $\rightarrow \mathbf{R} \setminus \{0\}$ is a weight assignment function on MODS with respect to s and **imp**, if for all $i \in$ MODS:

$$w(i) > 0 \Leftrightarrow i \ \textbf{imp} \ s \tag{7.1}$$

A weight assignment assigns a positive real number to each conforming implementation and a negative number to each erroneous implementation. For conforming implementations the weight can express that one implementation is better than another; negative weights express the gravity of errors in erroneous implementations: if $w(i_1) < w(i_2) < 0$ then both i_i and i_2 are not correct, but the errors of i_1 are more severe than those of i_2.

Example Let us consider the automaton *spec* of the specification from Figure 7.1. The specification has four states and the initial state is *I*. The set of inputs is $L_I = \{a\}$ and the set of outputs is $L_U = \{b, c\}$. In *I* the specification can receive the input a, produce the output b and after that c and arrive back in *I* or it can perform the c output. After performing c, the specification can receive a, produce c and arrive back in *I*.

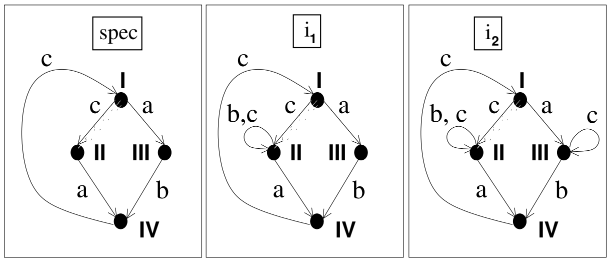


Figure 7.1: Example of a specification and two erroneous implementations.

In the same figure, two erroneous implementations i_1 and i_2 are shown. For the sake of simplicity we choose these implementations to not be IOTS (they are not used in the instantiation of this theory to *ioco* theory – see Section 7.4). Let us consider an arbitrary counting of bugs for the two implementations. The implementation i_1 has one bug: after c in the initial state, it can produce the unspecified outputs b and c. The implementation i_2 has more problems: it has the same bug as i_1 to which is added the bug of the state *III* where it can produce not only the output b but also the other output c. So intuitively the weight of i_1 could be $w(i_1) = -1$ and of i_2 could be $w(i_2) = -2$ because these implementations have one and two bugs, respectively.

Probability of implementations In conformance with assumption $\mathcal{A}$ an implementation i occurs with a probability $p_s(i)$. We consider MODS discrete (MODS must be finite or countably finite). Let $I \subseteq$ MODS, later to be used as follows: I is the set of all erroneous implementations. Then:

$$P_s(I) =_{\text{def}} \sum_{i \in I} p_s(i) \tag{7.2}$$

means the probability that the activity of implementing specification s produces an implementation that is modelled by a member of I.

Valuation Using the probability p_s and the weight assignment w it is possible to define a valuation on a discrete set of implementations (which gives the importance of a set of implementations I in terms of their weight and their probability of occurrence):

$$\mu(I) =_{\text{def}} \sum_{i \in I} w(i) p_s(i) \tag{7.3}$$

7.3 The coverage for on-the-fly test generation

Test generation and test run On-the-fly combines the test generation with the test run. For the remainder of this chapter we will abuse the words *test generation* for on-the-fly test generation and test run. Let $A \in$ ALGS be an algorithm for on-the-fly test generation. Let $\mathcal{O}$ be the set of possible test-run outcomes and let on-the-fly test generation and execution be correctly modelled by *genexec*: ALGS $\times$ SPECS $\times$ MODS $\rightarrow \mathcal{P}(\mathcal{O})$. Please note that $\mathcal{P}(\mathcal{O}) =$ OBS.

In accordance with assumption $\mathcal{B}$ running the algorithm A against an implementation i and using the specification s produces an outcome $\underline{\sigma}(A, s, i)$ from the set $genexec(A, s, i)$ with a probability $p_{A,s,i}(\sigma)$, with $\sigma = \underline{\sigma}(A, s, i)$. Note that the distribution of the variable $\underline{\sigma}(A, s, i)$ depends on the algorithm A, the specification s and the implementation i; for each A, s, i it may have another distribution. For a subset $O \subseteq genexec(A, s, i)$, the probability distribution is

$$P_{A,s,i}(O) =_{\text{def}} P(\underline{\sigma}(A, s, i) \in O) \tag{7.4}$$

The verdict for the observation will be **pass** if all outcome-verdicts are **pass**. Then let $v_t : \mathcal{O} \longrightarrow \{\textbf{pass},\textbf{fail}\}$ be a verdict assignment to outcomes, where t is a test generated with A ($t \in A(s, i)$); the probability measure $p_{A,s,i}$ can induce a probability measure on the set of verdicts $\{\textbf{pass},\textbf{fail}\}$. The probability that a single test generation of A with s and i results in a verdict **pass** is the cumulative probability that an outcome in $genexec(A, s, i)$ leads to a **pass** verdict.

$$\begin{aligned} p_{A,s,i}(\textbf{pass}) &=_{\text{def}} P(v_t(\underline{\sigma}(A, s, i)) = \textbf{pass}) \\ &= P_{A,s,i}(\{\sigma \in genexec(A, s, i) \mid v_t(\sigma) = \textbf{pass}\}) \end{aligned} \tag{7.5}$$

If the algorithm A will run more times against the implementation i, say m times, under the assumption that the test generations are independent, then the implementation passes all the test generations only if it passes all the individual test generations.

$$p^m_{A,s,i}(\textbf{pass}) = (p_{A,s,i}(\textbf{pass}))^m \tag{7.6}$$

The probability to **fail** is: $p^m_{A,s,i}(\textbf{fail}) =_{\text{def}} 1 - p^m_{A,s,i}(\textbf{pass})$. It can be shown that $p^m_{A,s,i}$ has the property that for non-zero probability of **fail** in a test generation, the algorithm will finally result in **fail** if enough test generations are performed (this property is expressed in the following proposition).

Proposition 7.3.1 *If* $p^1_{A,s,i}(\textbf{fail}) > 0$ *then* $\lim_{m\to\infty} p^m_{A,s,i}(\textbf{fail}) = 1$

For the proof see Appendix A.2.

The on-the-fly coverage The probability of occurrence of implementations was expressed by the probability measure P_s on the set of implementations (7.2). The probability that an implementation i yields the verdict **pass** with algorithm A was expressed by the probability measure $p_{A,s,i}$ (7.6). Under the assumption that the probability of occurrence of implementations is independent of the probability of yielding a **fail** we can integrate these measures to obtain the probability measure on MODS $\times\{\textbf{pass}, \textbf{fail}\}$. Taking also the weight of implementation into account we obtain analogously the valuation measure λ:

$$\lambda^m_{A,s}(I, V) = \sum_{i \in I} \sum_{v \in V} w(i) p^m_{A,s,i}(v) p_s(i) \tag{7.7}$$

Let $\overline{I_s}$ be the set of nonconforming implementations of s. We define the coverage of A, applied m times on a specification s, as

$$cov(A, s, m) =_{\text{def}} \frac{\lambda^m_{A,s}(\overline{I_s}, \{\textbf{fail}\})}{\lambda_{A,s}(\overline{I_s}, \{\textbf{pass}, \textbf{fail}\})} \tag{7.8}$$

where by $\lambda_{A,s}(\overline{I_s}, \{\textbf{pass}, \textbf{fail}\})$ we mean $\sum_{i \in \overline{I_s}} w(i) p_s(i)$ (we assume that an erroneous implementation occurs with a non-zero probability, formally $P_s(\overline{I_s}) > 0$).

In other words $cov(A, s, m)$ is the weighted probability of being able to conclude **fail** divided by the probability of an erroneous implementation to occur. The coverage is a function of the number of tests produced by the algorithm and it follows immediately from the definition that it has all the values in the range [0, 1].

Example So for example, if on average one out of three implementations is erroneous and if we assume $w(i) = -1$ for all erroneous implementations i, then $\lambda_{A,s}(\overline{I_s}, \{\textbf{pass}, \textbf{fail}\}) = -\frac{1}{3}$. Taking an arbitrary implementation and performing one test run will yield **fail** with a probability $\frac{1}{30}$, for example. So in $\frac{1}{3}$ of all cases we encounter an erroneous implementation and then we observe the bug in one out of ten cases on average. Then $\lambda^1_{A,s}(\overline{I_s}, \{\textbf{fail}\}) = -\frac{1}{30}$. So $cov(A, s, 1) = 0.1$, as expected. Moreover $\lambda^2_{A,s}(\overline{I_s}, \{\textbf{fail}\}) = -\frac{1}{3}(1 - (1 - \frac{1}{10})^2) \approx -0.063$. So $cov(A, s, 2) = \frac{-0.063}{-0.333} = 0.19$.

In this easy example we see that $cov(A, s, m)$ is monotonic in m and $\lim_{m \to \infty} cov(A, s, m) = 1$. As we will see, this holds in general under very reasonable conditions, which we set out to explore in this chapter. So for a non-zero distribution for $p^m_{A,s,i}$ and P_s the coverage has the monotonicity property. This property of the coverage is expressed in the following proposition.

Proposition 7.3.2 *Let $s \in$ SPECS be a specification and let $\overline{I_s}$ be the set of non-implementations of s. Let $A \in ALGS$ be an algorithm of on-the-fly test generation. Assume that an erroneous implementation occurs with a non-zero probability, formally $P_s(\overline{I_s}) > 0$. Assume that all faulty implementations that are possible can be detected, that is $\forall i \in \overline{I_s} : (p_s(i) > 0 \Rightarrow p_{A,s,i}(\textbf{fail}) > 0)$. Assume that $p_{A,s,i}(\textbf{fail}) < 1$. For positive integers m and n*

$$m < n \Rightarrow cov(A, s, m) < cov(A, s, n)$$

For the proof see Appendix A.3.

7.4 An application of the probabilistic coverage measure

In the previous sections we arrived at a coverage concept that applies to situations where we have a stochastic distribution of implementation errors and a stochastic, on-the-fly, test generation and execution algorithm. Assuming a relevance weighting for implementations we found that the coverage concept has nice properties, such as being in the range [0, 1] and being monotonic in the number of test runs. So far, the definitions are very abstract, making no assumptions about the nature of the specifications and implementations, or even about inputs and outputs.

Now we want to instantiate this with the *ioco* theory, which is described in Section 2.1. This is useful for two reasons. First, by instantiating our definitions we validate them. Secondly, the instantiated theory is relevant for the TorX algorithm and the TorX tooling. By working out a detailed example and choosing values for the probabilities we will obtain useful insights with respect to the working of TorX in practice. But in order to make the instantiation we have to introduce additional technicalities. First we need labelled transition systems based on a distinction between input labels and output labels, next to a null output. The implementation relation $\mathbf{ioco}_{\mathcal{F}}$ is parameterized over the set of traces $\mathcal{F}$, so we will choose a specific $\mathcal{F}$ in our running example. After that we can choose $p_s(i)$, the probability of implementations, the relevance weighting $w(i)$ and we instantiate A by a specific algorithm **Algo** (essentially this is TorX with certain probabilities).

Specifications The specification formalism SPECS is instantiated with the set of all transition systems over a label set L so we take SPECS= $\mathcal{LTS}(L_I \cup L_U)$. In Figure 7.1 we gave an example of a specification for $L = \{a\} \cup \{b, c\}$. Its suspension automaton is represented in Figure 7.2. It is easy to verify that the automaton from Figure 7.1 and the suspension automaton from Figure 7.2 have a similar structure. The only difference is that in $\{II\}$, the suspension automaton has a δ loop because no output is present in this state.

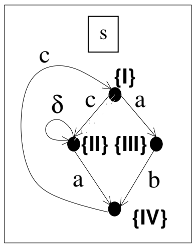


Figure 7.2: A suspension automaton.

Models of implementation We take MODS= $\mathcal{IOTS}(L_I, L_U)$. For the specification from Figure 7.1 the set of the inputs and of the outputs are the following: $L_I = \{a\}$ and $L_U = \{b, c\}$.

Tests We take TESTS= $\mathcal{TESTS}(L_I, L_U)$. Figure 7.3 shows the behaviour tree which contains all the execution traces of the tests derived with **Algo**, the algorithm which is to be described in the paragraph **Test generation** below.

Implementation relation We consider the $\mathbf{ioco}_{\mathcal{F}}$ implementation relation between SPECS and MODS. A good set of representative behaviours for $\mathcal{F}$ can be obtained by applying a test selection policy (see [CG96]). For our example we choose $\mathcal{F}$ to be the set formed by all the suspension traces which start from the initial state and which contain no cycle (they are cycling zero times via every state of the suspension automaton of the specification). So we put $\mathcal{F} = \{\epsilon, a, c, ca, ab\}$. We call the elements of $\mathcal{F}$ *representative behaviours*.

Definition 7.4.1 Let s be a specification, i an implementation and σ a suspension trace. If $out(i$ **after** $\sigma) \subseteq out(s$ **after** $\sigma)$ then we write $i \models_s \sigma$ and $i \not\models_s \sigma$ for its negation. If $i \not\models_s \sigma$ we say that i has fault σ with respect to s.

We take the same view as in [BTV91], that the existence of faults induces an equivalence relation

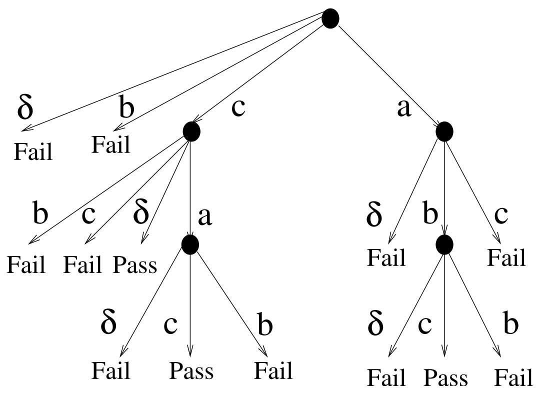


Figure 7.3: Tree containing the execution traces of all tests derived from the specification.

on the set of all implementations. Two implementations are equivalent if they contain the same faults. The equivalence relation induces a partition of the set of implementations MODS.

Please note that our set $\mathcal{F}$ is finite and consequently the number of equivalence classes is finite. Since $|\mathcal{F}|= 5$ there are 32 equivalence classes. Because in testing the objective is not to generate a model of the implementation under test but only to detect which faults are present in the implementation, it is valid to use the partition of MODS instead of MODS itself for the computation of the coverage.

Probability of implementations For a given specification s, let $p_\sigma(i)$ be the probability that an arbitrary implementation i has fault $\sigma \in \mathcal{F}$, under the assumption that the faults from each $\sigma \in \mathcal{F}$ are independent. Then we calculate

$$p_s(i) = \prod_{\sigma\in\mathcal{F}, i\models_s\sigma} (1 - p_\sigma(i)) \times \prod_{\sigma\in\mathcal{F}, i\not\models_s\sigma} p_\sigma(i) \tag{7.9}$$

which is a probability density function on $\mathcal{IOTS}(L_I, L_U)$.

Example The set of representative faults for our specification is given by the traces from $\mathcal{F} = \{\epsilon, a, c, ca, ab\}$. Then let the probability that an arbitrary implementation i violates the requirements $i \models_s \sigma$, for $\sigma \in \mathcal{F}$ be 0.2. The probability density function for our class of implementations is given by the following table (we will write σ for $i \models_s \sigma$ and $\neg\sigma$ for $i \not\models_s \sigma$, for $\sigma \in \mathcal{F}$):

Table 1

$p_s(i)$	ϵac	$\epsilon a\neg c$	$\epsilon\neg ac$	$\epsilon\neg a$ $\neg c$	$\neg\epsilon a$ c	$\neg\epsilon a$ $\neg c$	$\neg\epsilon\neg a$ c	$\neg\epsilon\neg a$ $\neg c$
$(ca)(ab)$	0.327	0.081	0.081	0.020	0.081	0.020	0.020	0.005
$(ca)\neg(ab)$	0.081	0.020	0.020	0.005	0.020	0.005	0.005	0.001
$\neg(ca)(ab)$	0.081	0.020	0.020	0.005	0.020	0.005	0.005	0.001
$\neg(ca)\neg(ab)$	0.020	0.005	0.005	0.001	0.005	0.001	0.001	0.0003

In the table the values of p_s are computed in a way similar as below:
$p_s((i \models_s \epsilon)(i \not\models_s a)(i \not\models_s c)(i \models_s ca)(i \models_s ab)) = (0.2)^2 \times (0.8)^3 = 0.020$

Weight of implementation A weight assignment for implementations can be obtained from weighing the faults that an implementation possesses. The function $w : \mathcal{IOTS}(L_I, L_U) \to \mathbf{R} \setminus \{0\}$ defined by:

$$w(i) = \begin{cases} 1 & \text{if } i \ \mathbf{ioco}_{\mathcal{F}} \ s \\ \sum_{\sigma \in \mathcal{F}, i \not\models_s \sigma} g(\sigma) & \text{otherwise} \end{cases} \tag{7.10}$$

where $g : \mathcal{F} \to \mathbf{R}_{<0}$ is a weight assignment function which expresses the gravity of violating $i \models_s \sigma$.

Example On the set of traces obtained by test selection $\mathcal{F} = \{\epsilon, a, c, ca, ab\}$, let $g(\epsilon) = g(a) = g(c) = g(ca) = g(ab) = -1$. Then it follows that $w(\epsilon ca(ca)(ab)) = 1, w(\epsilon \neg ca(ca)(ab)) = -1, w(\epsilon c \neg a(ca)(ab)) = -1, ..., w(\neg\epsilon\neg c\neg a\neg(ca)\neg(ab)) = -5$.

Test generation The algorithms from the universe ALGS will be described in natural language. Let us consider the *ioco* test generation algorithm from Section 2.1, which is the basis of the TorX algorithm. The on-the-fly algorithm is instantiated with the following implementation of the TorX specification, **Algo**, which works by applying these rules.

Algo

- if $\mathcal{F}$ equals $\emptyset$, Choice 1 of TorX (terminate the test case) has the probability 1 and Choice 2 and Choice 3 are in fact void (their probabilities are 0); in all other cases, Choice 1 is not taken (its probability p_1 is set to 0);
- if p_1 is 0: 1) if the specification is in a state which contains both inputs and outputs then the algorithm chooses Choice 2 and Choice 3 with equal probability ($p_2 = p_3 = \frac{1}{2}$, where $p_2 =_{\text{def}} P(\text{Choice 2}\}$, $p_3 =_{\text{def}} P(\text{Choice 3}\})$; 2) if the state of the specification contains only outputs then the algorithm chooses Choice 3 with probability one.

All the possible test execution traces generated with **Algo** for the suspension automaton s from Figure 7.1 and for initial $\mathcal{F} = \{\epsilon, a, c, ca, ab\}$ are represented in the behaviour tree from Figure 7.3. Now for a complete definition of *genexec* we need to define the test execution of a test t. This is

$$\text{exec}(t, i) =_{\text{def}} \{\sigma \mid \sigma \text{ is a test run of } t \text{ and } i\ \} \tag{7.11}$$

There are three possible tests which can be generated by **Algo**, which are $t_1 = \delta\ \mathbf{fail} + b\ \mathbf{fail} + c(b\ \mathbf{fail} + c\ \mathbf{fail} + \delta\ \mathbf{pass})$, $t_2 = \delta\ \mathbf{fail} + b\ \mathbf{fail} + ca(\delta\ \mathbf{fail} + b\ \mathbf{fail} + c\ \mathbf{pass})$ and $t_3 = a(\delta\ \mathbf{fail} + c\ \mathbf{fail} + b(\delta\ \mathbf{fail} + b\ \mathbf{fail} + c\ \mathbf{pass}))$ which form the set T. These tests are built in the following way. For example t_1 is built as follows: in the initial state **Algo** chooses to check the outputs; if the implementation sends a wrong output (δ or b) **Algo** finishes with a **fail** verdict; if the implementation sends the specified output c, **Algo** chooses to check the outputs; for the wrong outputs b and c **Algo** finishes with the **fail** verdict and for δ it finishes with the **pass** verdict. As we can see from t_1, **Algo** builds gradually these tests and by taking internal decisions. Sometimes it builds the test only partially. For example when the implementation sends a wrong output, such as b, **Algo** finishes with **fail** and it is not necessary to compute what follows after c; the partial test in this case is $t_p = \delta\ \mathbf{fail} + b\ \mathbf{fail} + ct'_p$. This partial test can be seen as the beginning of t_1 or t_2. Its observation is included in or equal to the observation of, for example, t_1 because every time when t_p is able to produce an outcome, t_1 is also able to produce it. This holds for every partial test: its observation is included in or equal to the observation of one of the three tests from T. So for the union of the observations of all possible tests produced by **Algo** when it runs against an implementation, which is the observation of **Algo** modelled by *genexec*, it is sufficient to consider the set T and so $genexec(\textbf{Algo},s, i) = \cup_{t \in T} exec(t, i)$.

Verdict assignment The verdict assignment function for a set of observations $O \subseteq genexec(\textbf{Algo}, s, i)$ is:

$$verd_T(O) = \begin{cases} \textbf{fail} & \text{if } \exists \sigma \in O, \exists t \in T : v_t(\sigma) = \textbf{fail} \\ \textbf{pass} & \text{otherwise} \end{cases} \tag{7.12}$$

where

$$v_t(\sigma) = \begin{cases} \textbf{pass} & \text{if } t \textbf{ after } \sigma = \{\textbf{pass}\} \\ \textbf{fail} & \text{if } t \textbf{ after } \sigma = \{\textbf{fail}\} \end{cases} \tag{7.13}$$

with $\sigma \in traces(t)$.

Probabilistic test generation and runs The probability that the specification of the TorX algorithm when it runs against an implementation stops at the end of a given trace is given by Lemma 5.2.1. For our instantiation **Algo**, we use the same formula as the one from Lemma 5.2.1, but with a small modification. The specification of TorX runs completely randomly; this means that the probability of Choice 2, p_2, and Choice 3, p_3, are global parameters. Our instantiation **Algo** which is especially adopted for our particular set of traces $\mathcal{F}$ and specification s has two different configurations for (p_2, p_3) when $p_1 = 0$: 1) $(\frac{1}{2}, \frac{1}{2})$ when **Algo** can send inputs and receive outputs (the current state of s contains inputs and outputs) and 2) $(0, 1)$ when **Algo** can only receive outputs (the current state of s contains only outputs). So in the formula from Theorem 5.2.1 we should consider the appropiate values of p_2 and p_3 in function of the possibility of **Algo** to send or not send inputs.

Example Figure 7.4 shows all the representative cases for the traces which end in a **fail** verdict that can be generated from the **Algo** algorithm when it runs against an implementation. There are five representative cases which correspond to the traces from $\mathcal{F} = \{\epsilon, a, c, ca, ab\}$, and these cases are A) $i \not\models_s \epsilon$; B) $i \not\models_s c$; C) $i \not\models_s a$; D) $i \not\models_s ca$; E) $i \not\models_s ab$. For computing the probability of **fail** for every particular class of implementation (as the class $(i \not\models_s \epsilon)(i \models_s a)(i \not\models_s c)(i \not\models_s ca)(i \models_s ab)$ for example) we should compute the probability of **fail** for every particular case.

Now, take for example the trace ca. The probability that **Algo** generates and runs this trace, so that it can detect the fault $i \not\models_s ca$, depends on the probability of trace c and consequently on the existence or non-existence of fault $i \not\models_s \epsilon$. If the implementation satisfies the requirement $i \models_s \epsilon$ then the only output which can be sent by i in the initial state is c ($P(i, c) = 1$); if the implementation violates the requirement $i \models_s \epsilon$, the implementation, in the initial state, can send c or another output, b or δ. So, for computing the probability of the trace ca we have the following representative combinations: D.1) $(i \models_s \epsilon)(i \not\models_s ca)$ (notation $\epsilon\neg(ca)$) and D.2)$(i \not\models_s \epsilon)(i \not\models_s ca)$ (notation $\neg\epsilon\neg(ca)$). In a similar way for the rest of the traces we have the following representative combinations: A) $(i \not\models_s \epsilon)$ (notation $\neg\epsilon$); B.1)$(i \models_s \epsilon)(i \not\models_s c)$ (notation $\epsilon\neg c$); B.2) $(i \not\models_s \epsilon)(i \not\models_s c)$ (notation $\neg\epsilon\neg c$); C) $(i \not\models_s a)$ (notation $\neg a$); E.1) $(i \models_s a)(i \not\models_s ab)$ (notation $a\neg(ab)$) and E.2) $(i \not\models_s a)(i \not\models_s ab)$ (notation $\neg a\neg(ab)$).

For computing the probability of **fail** it is necessary to compute the probability to arrive at the end of a fail-trace. For this we assume that if the implementation has a fault, then it can send (after

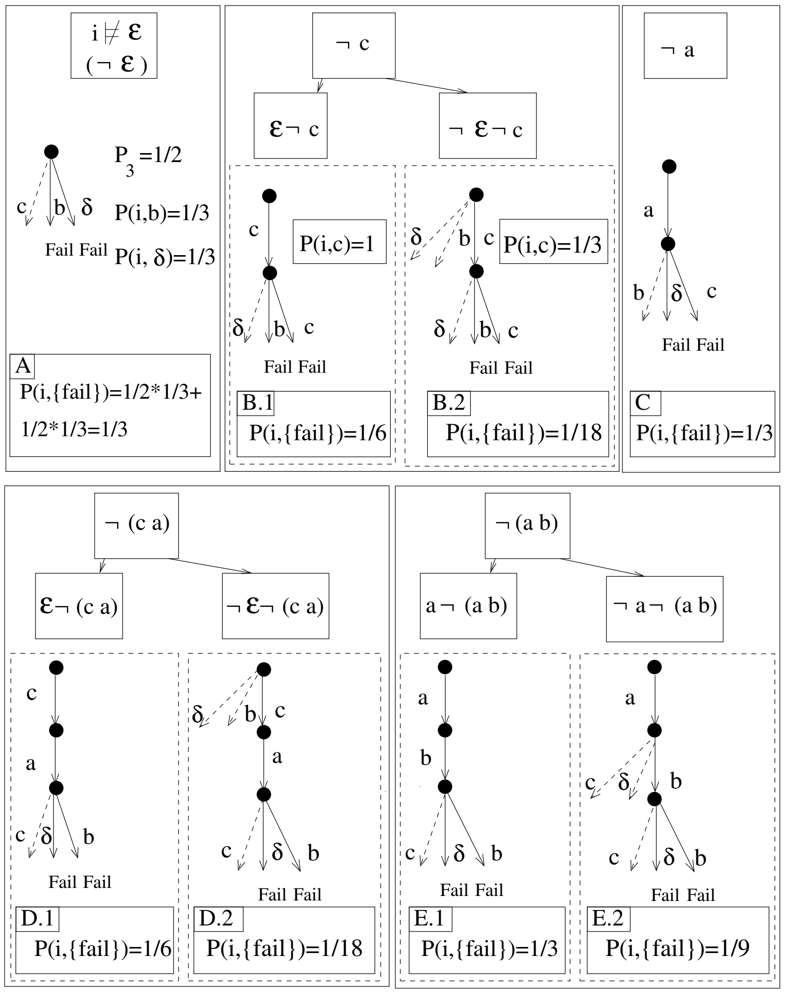


Figure 7.4: Representative cases for the traces which end with a **fail** verdict.

performing the trace correspondent to that fault) all the outputs with the same probability (uniform distribution for the outputs). So, for example, an implementation i which violates the requirement $i \models_s \epsilon$, sends the output δ with the same probability $\frac{1}{3}$ as for example the c signal (i in this case can send three possible outputs after computing ϵ trace, all of them with the same probability; so the probability to send one signal is $\frac{1}{3}$). Now, using this assumption, we will illustrate the way of computing the probability of **fail** for B.1)($i \models_s \epsilon$)($i \not\models_s c$) (notation $\epsilon\neg c$). In this case **Algo** gives a **fail** when it is performing the fail-traces cc or cb. Therefore the probability of **fail** is the sum of the probabilities of the fail-traces cc and cb. Please note that we computed the probability of **fail** for this particular representative combination (B.1) by summing the probabilities of all the fail-traces which have as root the trace of fault, c, to which is added an unspecified output, b or c (this results in the fail traces cb and cc). In a similar style we compute the probability of **fail** for the rest of representative combinations.

In the computation which follows by $p_{\mathbf{Algo},s,\epsilon\neg c}(\mathbf{fail})$ we mean the probability that **Algo** using the specification s and running against an implementation i which has the fault ($i \not\models_s c$) and respects the requirements ($i \models_s \epsilon$) fails when generating the fail-traces cc or cb; by $P(\epsilon\neg c$ **after** $\epsilon, c)$ we mean the probability that output c is sent by an implementation i from B.1 after performing the trace ϵ;

similar meanings have $P(\epsilon\neg c \textbf{ after } c, c)$ and $P(\epsilon\neg c \textbf{ after } c, b)$. In accordance to Lemma 5.2.1 the computation is:

$$p_{\textbf{Algo},s,\epsilon\neg c}(\textbf{fail}) = P(\text{Choice 3}) \times P(\epsilon\neg c \textbf{ after } \epsilon, c)\times$$

$$\times(P(\text{Choice 3}) \times P(\epsilon\neg c \textbf{ after } c, c) + P(\text{Choice 3}) \times P(\epsilon\neg c \textbf{ after } c, b)) =$$

$$= \frac{1}{2} \times 1 \times (\frac{1}{2} \times \frac{1}{3} + \frac{1}{2} \times \frac{1}{3}) = \frac{1}{6}$$

The probabilities of **fail** for the rest of the cases are :

A) $p_{\textbf{Algo},s,\neg\epsilon}(\textbf{fail}) = \frac{1}{3}$;
B.2) $p_{\textbf{Algo},s,\neg\epsilon\neg c}(\textbf{fail}) = \frac{1}{18}$;
C) $p_{\textbf{Algo},s,\neg a}(\textbf{fail}) = \frac{1}{3}$;
D.1) $p_{\textbf{Algo},s,\epsilon\neg(ca)}(\textbf{fail}) = \frac{1}{6}$; D.2) $p_{\textbf{Algo},s,\neg\epsilon\neg(ca)}(\textbf{fail}) = \frac{1}{18}$;
E.1) $p_{\textbf{Algo},s,a\neg(ab)}(\textbf{fail}) = \frac{1}{3}$; E.2) $p_{\textbf{Algo},s,\neg a\neg(ab)}(\textbf{fail}) = \frac{1}{9}$.

Using these results, the probability of **fail** for every particular class of implementations is easily computed by summing the probability of **fail** of its component representative combinations. So take as an example the class $(i \not\models_s \epsilon)(i \models_s a)(i \not\models_s c)(i \not\models_s ca)(i \models_s ab)$ (notation $\neg\epsilon a\neg c\neg(ca)(ab)$):

$$p_{\textbf{Algo},s,\neg\epsilon a\neg c\neg(ca)(ab)}(\textbf{fail}) = p_{\textbf{Algo},s,\neg\epsilon}(\textbf{fail}) + p_{\textbf{Algo},s,\neg\epsilon\neg c}(\textbf{fail}) + p_{\textbf{Algo},s,\neg\epsilon\neg(ca)}(\textbf{fail})$$

$$= \frac{1}{3} + \frac{1}{18} + \frac{1}{18} = \frac{4}{9} = 0.44$$

Please note that we replace the implementation from $p_{A,s,i}$ by its correspondent class. The probabilities of **fail** for every class of implementation i are given in the following table.

Table 2

$p_{\textbf{Algo},s,i}$ (**fail**)	ϵac	$\epsilon a\neg c$	$\epsilon\neg ac$	$\epsilon\neg a$ $\neg c$	$\neg\epsilon a$ c	$\neg\epsilon a$ $\neg c$	$\neg\epsilon\neg a$ c	$\neg\epsilon\neg a$ $\neg c$
$(ca)(ab)$	0	0.16	0.33	0.49	0.33	0.38	0.66	0.71
$(ca)\neg(ab)$	0.33	0.49	0.44	0.60	0.66	0.71	0.77	0.82
$\neg(ca)(ab)$	0.16	0.32	0.49	0.64	0.38	0.44	0.71	0.76
$\neg(ca)\neg(ab)$	0.49	0.65	0.60	0.76	0.71	0.76	0.82	0.87

The probability density function $p_s(i)$ was given in *Table 1* for every particular class of implementation. The probabilities of **fail** when **Algo** runs one time against an implementation were given in *Table 2* for every particular class of implementation. Then the coverage of the **Algo** algorithm when it runs m times against an implementation is:

$$cov(\textbf{Algo}, s, m) =_{\text{def}} \frac{\lambda^m_{A,s}(\overline{I_s}, \{\textbf{fail}\})}{\lambda_{A,s}(\overline{I_s}, \{\textbf{pass}, \textbf{fail}\})} = \frac{\sum_{i\notin I_s} w(i)p^m_{\textbf{Algo},s,i}(\textbf{fail})p_s(i)}{\sum_{i\notin I_s} w(i)p_s(i)} =$$

$$\frac{w(\epsilon a\neg c(ca)(cb))p^m_{\textbf{Algo},s,\epsilon a\neg c(ca)(cb)}(\textbf{fail})p_s(\epsilon a\neg c(ca)(cb) + ...}{w(\epsilon a\neg c(ca)(cb))p_s(\epsilon a\neg c(ca)(cb) + ...}$$

$$\frac{... + w(\neg\epsilon\neg a\neg c\neg(ca)\neg(cb))p^m_{\textbf{Algo},s,\neg\epsilon\neg a\neg c\neg(ca)\neg(cb)}(\textbf{fail})p_s(\neg\epsilon\neg a\neg c\neg(ca)\neg(cb)}{... + w(\neg\epsilon\neg a\neg c\neg(ca)\neg(cb))p_s(\neg\epsilon\neg a\neg c\neg(ca)\neg(cb)} =$$

$$\frac{-1 \times (1-(1-0.16)^m) \times 0.081 - \ldots - 5 \times (1-(1-0.87)^m) \times 0.0003}{-1 \times 0.081 - \ldots - 5 \times 0.0003}$$

In the formula of the coverage from above there are 31 terms corresponding to the erroneous classes of implementations. Because, there are too many terms to be represented, we consider in the formula only the first and the last terms and we skip the 29 intermediary terms. For the computation of the coverage we made a small program which computed it for different values of m. So the points are $cov(\mathbf{Algo}, s, 1) = 0.42$, $cov(\mathbf{Algo}, s, 2) = 0.63$, $cov(\mathbf{Algo}, s, 3) = 0.75$, $cov(\mathbf{Algo}, s, 4) = 0.82$... For the limit case we find $\lim_{m\to\infty} cov(\mathbf{Algo}, s, m) = 1$.

From the computation we see that after running **Algo** once we obtain a coverage of almost one half, running twice, it covers two thirds and running it many times it has the maximum coverage 1. Looking at the size of the specification which has 4 states and 6 transitions and of $\mathcal{F}$ which consists of 5 traces we can expect that the coverage should go rapidly to the limit one; for example a deterministic algorithm after five runs, on average, is expected to check all the traces of $\mathcal{F}$. This expectation is also confirmed by the observation that an erroneous implementation can easily occur and **Algo** can easily detect an error (the probability of obtaining a **fail** is in average 0.5 for a run of the algorithm). This property is shown by the computation of the coverage which has after four runs a high coverage of 0.82 and after one run a coverage of almost 0.5.

It is interesting to see how changes of some parameters of the coverage formula influence the coverage values. For example, what happens when **Algo** does not detect an error so easily? Let us decrease the values of the probabilities of obtaining a **fail** with **Algo** from *Table 2* with a factor of $\frac{1}{10}$. In this case the probability of obtaining a **fail** is on average 0.05 for a run of the algorithm. The coverage computation gives the following values: $cov(\mathbf{Algo}, s, 1) = 0.042$, $cov(\mathbf{Algo}, s, 2) = 0.082$, $cov(\mathbf{Algo}, s, 3) = 0.12$, $cov(\mathbf{Algo}, s, 4) = 0.15$... The values express the expected property that when it is difficult to detect an error, the coverage does not increase quickly to one. To complete the picture we should say that when modifying the weight of an arbitrarily chosen fault, $\neg(ab)$, by increasing it to say 10, the coverage values increase by about 5%. This can be expected because the coverage follows the behaviours of the erroneous implementations which contain the fault $\neg(ab)$ and which have a higher average, of about 0.6, for the probability of obtaining a fault.

7.5 Conclusions

This chapter extends the work from [BTV91, HT96] and [FGM00] with a coverage measure for an algorithm which generates and runs tests on-the-fly. The probability of sending outputs by the implementation and consequently of obtaining a verdict by the on-the-fly algorithm reflects the probabilistic nature of this coverage. The severity of the bugs in the erroneous implementation, the probability distribution of the implementations and the probability distribution of producing a **fail** by the on-the-fly algorithm are combined in the probabilistic coverage.

This theory is instantiated for the *ioco* theory and the TorX algorithm and an example for this instantiation is worked out. When the number of test runs increases, the coverage increases arriving at the limit one, provided that there is a non-zero probability of **fail**. This expresses the expected property that after performing a sufficient quantity of test runs, the algorithm is able to detect at the end all bugs of an erroneous implementation. When decreasing the probability of detecting an error, the coverage also decreases, as expected. When increasing the weight of an arbitrary fault, the coverage follows the behaviours of the erroneous implementations which do contain the fault. Computer programs can be made for the computation of the coverage, as a part of test generation tools (such as TorX).

Chapter 5, 6 and 7 present ways of controlling the on-the-fly test generation and execution and ways of defining coverage measures for on-the-fly algorithms. Chapter 8 deals with another topic, namely test selection. By applying test selection a reduced set of tests is selected. Chapter 8 presents a coverage measure which expresses the error detection power of a reduced set of tests. We should note that there is no conflict between the two coverages formalized, the one for the on-the-fly test generation and execution, from Chapter 7, and the one for test selection, from Chapter 8. As explained in Chapter 1, the test selection is done before any test generation and execution. An intuitive example of a selection is to limit some parameter ranges of signals (from a specification which has signals with parameters, such as the Conference Protocol specification) to a small number of values. The coverage for test selection, from Chapter 8, expresses the detection power of the reduced set of tests which is chosen by selection. This detection power is compared to the set of errors which can be discovered by the whole set of tests. Now, from the set of tests selected only some tests are generated and executed by an on-the-fly algorithm (not necessarily all of them). The coverage for the test generation and execution expresses the detection power of the test suite which is generated and executed. In this case, the detection power is compared to the set of errors which can be discovered by the reduced set of tests. The two values obtained corresponding to the two coverages can be combined (for example by multiplying them) to express the general detection power of the test suite which resulted after selection, generation and execution. The general detection power of the test suite is compared to the set of errors which can be discovered by whole set of tests. In this way the two coverages formalized in Chapter 7 and in Chapter 8 complement each other.

Chapter 8

Test Selection, Trace Distance and Heuristics

8.1 Introduction

Testing provides developers, users, and purchasers, with increased levels of confidence in the product quality. Conformance tests capture the behavioural description of a specification and measure whether a product, or IUT (the implementation under test), faithfully implements the specification. Because of time and resource limitations, any form of testing can only exercise a small subset of all possible system behaviours. Therefore, testing can never give certainty about the correctness of a system.

Since in practice exhaustive testing is impossible, an important step in the testing process is the development of a carefully selected test suite, i.e., a set of test cases. Such a test suite should have a large potential of revealing errors in the implementation. Moreover, we would like to be able to compare different test suites in order to select the best one, and to quantify their error-detecting capability.

The selection of an appropriate set of tests from all possible ones (usually infinitely many test cases), is not a trivial task. We refer to this task as *test selection*. Traditionally, test selection is based on a number of heuristic criteria. Well-known heuristics include equivalence partitioning, boundary value analysis, and use of code-coverage criteria like statement-, decision- and path-coverage [Mye79]. Although these criteria provide some heuristics for selecting test cases, they are rather informal and they do not allow to measure the error-detecting capability of a test suite.

If test cases are derived from a formal specification, in particular if it is done algorithmically using tools for automatic test generation, e.g., Autolink [SKGH97], TGV [JM99] or TorX [BFdV^{+}99], then the test selection problem is even more apparent. These test tools can generate a large number of test cases, when given a specification in the appropriate formalism, without much user intervention. All these generated test cases can detect potential errors in implementations, and errors detected with these test cases indeed indicate that an implementation is not correct with respect to its specification. However, the number of potentially generated test cases may be very large, or even infinite. In order to control and get insight in the selection of the tests, and by that get confidence in the correctness of an IUT that passes the tests, it is important that the selection process is formally described and based on a well-defined strategy.

It should be noted, however, that test selection is an activity that in principle cannot be based solely on a formal specification of a system. In order to decide which test cases are more valuable than others, either extra information outside the realm of the specification formalism is necessary, or assumptions about the occurrence of errors in the implementation must be made. Such extra information may include knowledge about which errors are frequently made by implementers, which kind of errors are important, e.g., in the sense of having catastrophic consequences, what functionality is difficult to implement, which functionality is crucial for the well functioning of the system, etc. An approach to formalizing this extra information was given in [BTV91]. On the other hand, assumptions can be made about the occurrence of errors in implementations, e.g., that errors will not occur in isolation, i.e., if some behaviour is erroneous then there is a large probability that some other behaviour close to it is also erroneous. So we only have to test one of these behaviours (equivalence partitioning: the behaviours are equivalent with respect to the occurrence of errors). Another often used assumption is that errors are most likely to occur on the boundaries of valid data intervals (boundary value analysis).

We approach the problem of test selection by making assumptions in an automata-based, or labelled transition system-based formalism. The results of the research described in this chapter were also presented in [FGMT02]. Up to this point, the theory presented in this thesis was based on the *ioco* theory which is depicted in Chapter 2. For the theory of test selection not all the ingredients of *ioco* are needed. Therefore we recall in Section 8.2 the basic definitions which are used for the test selection formalization. Two different kinds of assumptions are introduced and expressed as *heuristic principles* in Section 8.3 starting with the ideas of [CG96]. The first one, called *reduction heuristic*, assumes that few outgoing transitions of a state show essentially different behaviour. The second one, referred to as *cycling heuristic*, assumes that the probability to detect erroneous behaviour in a loop decreases after each correct execution of the loop behaviour. After that we propose a mathematical framework, defining a heuristic as a function on the set of behaviours (traces). This is done in Section 8.4. When we want to make the two heuristics more precise, defining them as functions according to the definition from Section 8.4, we observe that an appropriate behaviour representation for them is needed. Therefore in Section 8.5 we define the *marked trace* representation. After these preparations the definitions of the heuristics as functions on marked traces are straightforward (Section 8.6). Subsequently, the notion of isolation and closeness of errors is formalized in Section 8.7 by defining a *distance* function between behaviours. This idea is taken from [ACV93, ACV97] and extended to *marked traces*. The trace distance implements the considered heuristics in the sense that the traces which are selected by the heuristics are remote from each other. Every trace which is excluded by the heuristics is close to one of the selected traces. A *coverage function* which may serve as a measure for the error-detecting capability of a test suite is defined based on the maximum distance between selected and non-selected behaviours and a formula for approximating the coverage is given in Section 8.8.

8.2 Preliminaries

The basic formalism for our discussion about test selection is the labelled transition system, or the automaton. A labelled transition system provides means to specify, model, analyze and reason about (concurrent) system behaviour. A labelled transition system is defined in terms of states and labelled transitions between states. We recall Definition 2.1.1 for the definition of a labelled transition system.

The labels in L represent the actions of a system. An action $a \in L$ is executable in state $q \in Q$ if $(q, a, q') \in T$ for some state $q' \in Q$, which is said to be the new state after execution of a; we also write $q \xrightarrow{a} q'$. A finite sequence of pairs *state, action* ending into a state is called a *path*, i.e. a sequence of type $q_1 a_1 q_2 .... q_n a_n q_{n+1}$ with $n \in \mathbf{N}$, $i \leq n+1$, $q_i \in Q$, $a_i \in L$ and $q_1 \xrightarrow{a_1} \ldots \xrightarrow{a_n} q_{n+1}$. Similarly, a finite sequence of actions is called a *trace*. The set of all traces over L is denoted by L^*, with ϵ denoting the empty sequence. Abusing notation, we will use p to denote both the labelled transition system and the current (or initial) state of the system.

The traces of a labelled transition system p are all sequences of actions that p can execute from its initial state q_0: $traces(p) =_{\text{def}} \{ \sigma \in L^* \mid q_0 \xrightarrow{\sigma} \}$. Here we use the following additional definitions ($n \in \mathbf{N}, i \leq n, q, q', q_i \in Q, a_i \in L, \sigma \in L^*$):

$$
\begin{array}{lll}
q \xrightarrow{a_1 \cdot \ldots \cdot a_n} q' & =_{\text{def}} & \exists q_0, \ldots, q_n : q = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n = q' \\
q \xrightarrow{\sigma} & =_{\text{def}} & \exists q' : q \xrightarrow{\sigma} q'
\end{array}
$$

For our presentation and formalization we use *minimal*, *deterministic*, *finite-state* transition systems. A finite-state labelled transition system has a finite number of states, i.e., Q is finite. A transition system is deterministic if for any state $q \in Q$ and action $a \in L$ there is at most one successor state, i.e., $T : Q \times L \to Q$ is a (partial) function. A transition system is minimal if there are no equivalent states, i.e., no two states with exactly the same traces, which means: $\nexists q, q' \in Q, q \neq q' : traces(q) = traces(q')$. We (ab)use the word *automaton* for these minimal, deterministic, finite-state transition systems.

Although it may seem a severe limitation to restrict to automata, an important formal test theory, viz. **ioco**-testing [Tre96], can be expressed, in terms of so-called *suspension automata*. So the test selection approach which is presented in this chapter can be integrated with **ioco**-testing.

In testing, the traces of the minimal, deterministic, finite automata are used. A complete (maximal) test suite for an automaton specification s is expressed as $traces(s)$. However, even if s is finite-state, its set of traces will usually be infinite and contain traces of unbounded length. Hence, a complete test suite will have infinitely many tests of unbounded length. Such a test suite can never be executed within any reasonable limits of time and resources. Consequently, the problem of *test selection* consists of selecting a finite subset $T \subseteq traces(s)$, such that we end up with a reasonably sized set of bounded-length test cases.

The challenge of test selection now is to choose T such that the resulting test suite keeps a large error-detecting capability. Moreover, we wish to quantify this capability in order to compare and select test suites. The next sections will present and formalize an approach to selection and quantification.

8.3 Introduction to heuristics, distance and coverage

In this section we introduce the concepts of heuristics and coverage. Two specific heuristics will be proposed in Section 8.3.1. They are illustrated by an example in Section 8.3.2.

8.3.1 The heuristics principles for the test selection

As motivated in Section 8.2, the specification is seen as a minimal finite-state automaton. The specification has a set of traces which usually is too large; for this reason, we want to obtain a smaller set of traces. As explained in Section 8.1, this goal can be reached by making assumptions on the occurrence of faults, assumptions which are expressed as heuristic principles. The heuristic principles with which we are working in this chapter are:

- *Reduction:* if the specification automaton contains for a state a large number of outgoing transitions which go to the same next state, only a small number of these transitions need to be selected;

- *Cycling:* each cycle in the automaton needs to be traversed only a limited number of times by every single trace.

8.3.2 A first illustration of the test selection

Now we will give an example on which we will apply our heuristics for test selection.

Let s be the specification automaton from the left-hand side of Figure 8.1. The specification has four states. The labelset is $L = \{b, c, d, e, f\} \cup \{a_i \mid i \in \mathbb{N}\}$ and the initial state is the state *I*. This state has infinitely many outgoing transitions ($\{a_i \mid i \in \mathbb{N}\}$). Via a transition a_i from the initial state, one arrives at *II*. This state contains a cycle which goes via *III* using the transitions b and d; the state *III* contains another cycle, via the transition c. From *II* one arrives at *IV* using e or f. For simplicity, we will consider only the traces of s that end in *IV*. Let T be this set of traces of s.

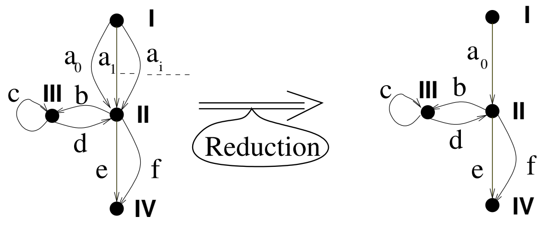


Figure 8.1: A minimal automaton of a specification and its reduced version.

Now let us consider a_0 as being the representative transition in the initial state; by choosing this transition the *Reduction* heuristic is applied in this state. By this application we reduce the labelset to a finite one $L' = \{a_0, b, c, d, e, f\}$. This labelset L' corresponds to the reduced automaton from the right-hand side of Figure 8.1. Now our initial set of traces becomes $T^{Reduction}$ and it contains all the traces of the automaton which are starting from state *I*, arriving in state *IV* and going through transition a_0 in *I* ($T^{Reduction}$ equals also the set of traces of the reduced automaton). In this example one representative is selected; in a more general example it could also be two or more of the a_i.

The following heuristic to be applied is the *Cycling* heuristic. The traces are cycling via the states *II* and *III* of the automaton. In this automaton the state *II* has a cycle via the sequence of transitions bd and the state *III* has another cycle via the transition c. We can fix the cycle limit number to 1 for the cycling states *II* and *III*. So the transitions c from *III* and bd from *II* can be traversed only once by every single trace of $T^{Reduction}$. The traces which respect this condition, and thus form the set of traces $T^{Reduction,\ Cycling}$, are:

$$T^{Reduction,\ Cycling} = \{a_0e, a_0bde, a_0bcde, a_0f, a_0bdf, a_0bcdf\}$$

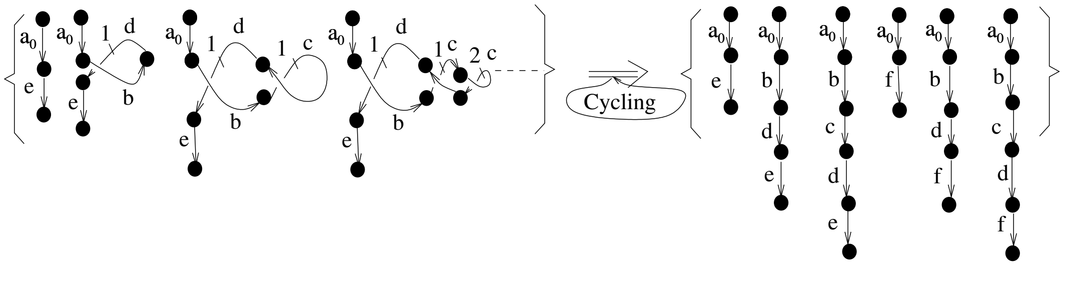


Figure 8.2: An application of the *Cycling* heuristic.

The application of the *Cycling* heuristic is represented graphically in Figure 8.2. The full set is represented at the left-hand side and the reduced set at the right. As it can be seen the set $T^{Reduction,\ Cycling}$

is a finite set and all its traces have a length of at most 5 (finite length).

Our method deals with bigger cycles as well, such as going via the transitions *bd* several times; the technique of [ACV93] deals only with simple cycles – such as going via transition *c* several times. Another advantage is that with the proposed test selection technique one can deal with an infinite branching of transitions (see the initial state of this automaton). As we saw in our example, limiting the cycle number implicitly limits the length if the automaton is finite and therefore a length heuristic, which is considered by [ACV93], is not necessary here.

Now we are going to express the heuristic principles in terms of distances among traces. A distance is a measure which expresses how far apart two traces are. A particular way to compute such a trace distance is given in Section 8.7.2. To get a feeling of how the trace distance is related to the heuristic principles, let us take as an example the distance between the traces a_0bdf and a_0bdbdf. In Figure 8.3, it can be seen that the distance between the traces a_0bdf and a_0bdbdf is smaller than the distance for example between a_0f and a_0bdbdf. This happens because the trace a_0bdf cycles one time via the state *II*, a_0bdbdf cycles twice and a_0f cycles zero times. Therefore intuitively, the trace a_0bdf should be closer to a_0bdbdf than to the other traces (exactly as we assume in the *Cycling* heuristic that the later cycles are less important, so the distance between two traces which are cycling more often through a state will decrease).

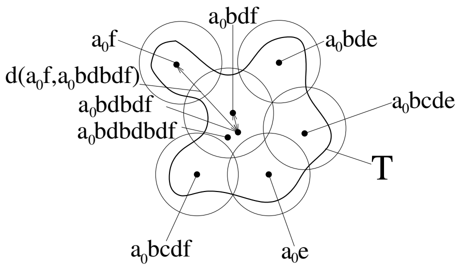


Figure 8.3: A covering of the initial set using trace distance.

In Figure 8.3, every trace from the reduced set $T^{\textit{Reduction, Cycling}}$ is the center of a sphere. The initial set T is covered by the reduced set $T^{\textit{Reduction, Cycling}}$, such that every trace from T has a corresponding trace in $T^{\textit{Reduction, Cycling}}$ to which the distance is smaller than a given limit ε (ε is the radius of the spheres). This process of selecting one representative for each sphere leads to a notion of coverage. When taking big spheres only few representatives are selected and the error detection capability is low. Small spheres, on the other hand give a large coverage. If we scale things in such a way that $0 \leq \varepsilon \leq 1$, then the coverage can be expressed as $1-\varepsilon$. The coverage of the reduction from T to $T^{\textit{Reduction, Cycling}}$ is denoted as $cov(T^{\textit{Reduction, Cycling}}, T)$. Therefore we express $cov(T^{\textit{Reduction, Cycling}}, T) = 1-\varepsilon$.

This gave some intuition about how the heuristics and the trace distance are used in the test selection and computation of the coverage. In the following section we are going to be more formal.

8.4 The trace distance and the test heuristics

In our test selection method we use heuristics which are applied to traces and distances between traces. This section describes the formal definitions of these notions.

Formally, a trace heuristic is a function between two sets of traces such that the range is a proper subset of the domain (so the heuristic reduces the size of the initial set).

Definition 8.4.1 A trace heuristic h is a function $h : T \to T$, where T is a set of traces and $Ran(h) \subset T$.

Definition 8.4.2 Let T be a set. Then a function $d : T \times T \to \mathbf{R}_{\geq 0}$ is a distance iff: 1) $d(x, x) = 0$; 2) $d(x, y) = d(y, x)$; 3) $d(x, y) \leq d(x, z) + d(z, y)$; for all $x, y, z \in T$.

In particular we use Definition 8.4.2 for sets of traces and such distances are called trace distances. The pair (T, d) is a metric space. A space where d does not necessarily satisfy the triangle law (expressed by the point 3) from the definition above) is called a semi-metric space, or sometimes for simplification, still a metric space. We can weaken the conditions in which this theory applies for semi-metric spaces because the triangle law property is not really needed (used) for the theory presented here. In the remainder of this chapter we are using 'classic' metric spaces which have also the triangle property. It is customary to express coverages by numbers in the range [0, 1] and therefore we restrict ourselves to distance functions such that $0 \leq d(x, y) \leq 1$ for all x, y. This can be done without loss of generality (suppose we would have distances in $[0, \infty]$ and ε numbers (we will come later to them) in the range $[0, \infty]$ then we could scale them back to [0, 1] using a suitable monotonic and continuous bijection $b : [0, \infty] \to [0, 1]$). In order to use a trace distance for test selection the concept of ε-cover is useful.

Definition 8.4.3 A set T' is an ε-cover of T ($T' \subseteq T, \varepsilon \geq 0$) with respect to distance d if for every $t \in T$ there exists $t' \in T'$ such that $d(t, t') \leq \varepsilon$.

The concept of ε-cover gives rise to the property of total boundedness for a metric space.

Definition 8.4.4 A metric space (T, d) is totally bounded if for every $\varepsilon > 0$ it is possible to find a finite set $T_\varepsilon \subseteq T$ such that T_ε is an ε-cover of T with respect to distance d.

Now a link between a heuristic and a trace distance is established: if for a given heuristic the subset obtained by the application of the heuristic is an ε-cover of the original set, then the trace distance implements the heuristic.

Definition 8.4.5 Let T be a set of traces and h be a trace heuristic such that $h : T \to T$. Let d be a trace distance defined on T. Then d implements the heuristic h iff: $\exists \varepsilon_h \geq 0 : Ran(h)$ is an ε_h-cover of T with respect to the distance d.

The following definition shows how to obtain the coverage.

Definition 8.4.6 Let T be a set of traces and $T' \subseteq T$ be a cover of T with respect to a trace distance d. Let $\varepsilon_m = inf\{\varepsilon \geq 0 \mid T' \text{ is an } \varepsilon\text{-cover of } T\}$ be the inferior minimum of the ε values. Then the coverage of T' with respect to T is $cov(T', T) = 1 - \varepsilon_m$.

8.5 The marked trace representation

When we want to make the two heuristics more precise, defining them as functions according to Definition 8.4.1, we observe that an appropriate trace representation for them is needed. When we apply the *Cycling* heuristic to a trace, we observe that the trace does not have enough information regarding how it was generated, what states it has been going through and how often it went through each state. As a result, we will represent the trace in such a way that the information regarding its generation from the automaton will be included. This leads us to a concept called marked traces, which will be developed in Section 8.5.1. In general a given trace can be interpreted in several ways as being the result of running through cycles in the automaton. This introduces a problem of ambiguity which is addressed in Section 8.5.2.

8.5.1 The marked traces

The first example in this section will explain why a new representation for the traces is needed.

Example Let us consider the automaton from Figure 8.4 and one of its traces, $abcbcd$. This trace is traversing (cycling) twice via the state II. But the state information is not present in the trace. Therefore this representation is not appropriate for working with cycles. Now let us transform it into a path, which is $IaIIbIIIcIIbIIIcIIdIV$. We can observe that the path contains extra information which is not needed for cycles: for example it contains the states I and IV which are not part of any cycle. Summing up the observations, we arrive at the conclusion that a new representation is needed. An intuitive one is $a[\langle bc\rangle\langle bc\rangle]^{II}d$ where $[\langle bc\rangle\langle bc\rangle]^{II}$ indicates that two cycles consisting of the transitions bc are performed through the state II.

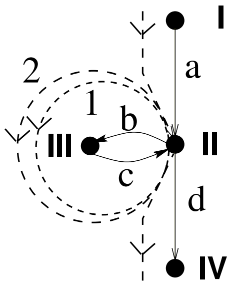


Figure 8.4: A trace which cycles through an automaton.

As we saw in the introductory example, we associate the cycles with how many times a trace is traversing a state. The name of the state, which is seen as a mark, will serve as the identifier of the cycle. We call such an extended trace a *marked trace*. Now we have all the ingredients to define a marked representation of a trace. Let L be a labelset and Q a set of states (or marks). We will define below the grammar G which generates marked traces.

Grammar (G)

non-terminals:

mt	(marked trace)
nemt	(non-empty marked trace)
neseq	(non-empty sequence)

terminals:

u	$(u \in L)$
q	$(q \in Q)$
$[,], \langle, \rangle$	(two types of brackets)

rules:

$$\begin{array}{llll} \text{mt} & \rightarrow & \epsilon & \\ & \mid & \text{nemt} & \\ & & & \\ \text{nemt} & \rightarrow & u \text{ mt} & (u \in L) \\ & \mid & [\text{ neseq }]^{q} \text{ mt} & (q \in Q) \\ & & & \\ \text{neseq} & \rightarrow & \langle \text{ nemt } \rangle & \\ & \mid & \langle \text{ nemt } \rangle \text{ neseq} & \end{array}$$

start symbol:
mt

It is easy to check that the grammar G is not ambiguous. Below, the definition for the marked traces is given.

Definition 8.5.1 A marked trace is an element of the language of the grammar G, i.e., a sequence generated by the grammar G.

Example Some examples of marked traces are: $a[\langle bc\rangle]^{II}d$ and $a[\langle bc\rangle\langle bc\rangle]^{II}d$ with $II \in Q$.

We will denote the set of all the marked traces over a labelset L and a set of marks Q as L^*_Q.

The transformation between the marked representation of a trace and a normal representation of a trace can be made easily by eliminating all the parentheses and states which occur in the marked representation. For example the marked trace $a[\langle bc\rangle\langle bc\rangle]^{II}d$ is transformed in the trace $abcbcd$. We will call this transformation *unfold*.

Definition 8.5.2 Let L be a labelset, let Q be the set of marks and let L^*_Q be the set of marked traces. Then the function $L^*_Q \rightarrow L^*$ which transforms a marked trace into a trace is:

1. if t is a marked trace because t is ϵ then $\mathit{unfold}(t) = \epsilon$;
2. if t is a marked trace because t is t' with t' a non-empty marked trace then $\mathit{unfold}(t) = \mathit{unfold}(t')$;
3. if t is a non-empty marked trace because t is ut' with $u \in L$ and t' a marked trace then $\mathit{unfold}(t) = u\ \mathit{unfold}(t')$;
4. if t is a non-empty marked trace because t is $[s]^q t'$ with s a non-empty sequence, $q \in Q$ and t' a marked trace then $\mathit{unfold}(t) = \mathit{unfold}(s)\mathit{unfold}(t')$;
5. if s is a non-empty sequence because s is $\langle t\rangle$ with t a non-empty marked trace then $\mathit{unfold}(s) = \mathit{unfold}(t)$;

6. if s is a non-empty sequence because s is $\langle t\rangle s'$ with t a non-empty marked trace and s' a non-empty sequence then $unfold(s) = unfold(t)unfold(s')$.

In the following example, we will illustrate a way in which a trace can be transformed into a marked trace. In general, this transformation is not unique. To illustrate this, we need a more complex example.

Example Consider the automaton from Figure 8.1. The states of the automaton are marked with *I, II, III, IV*. The cycling states are the states *II* and *III*. Consider the trace $a_0bcdbcde$. Adding boxes to reflect the nesting structure, the corresponding path is $Ia_0\boxed{II}b\boxed{\boxed{III}}c\boxed{\boxed{III}}d\boxed{II}b\boxed{\boxed{III}}c\boxed{\boxed{III}}d\boxed{II}eIV$. The state *II* (surrounded with a single box in the path) appears three times. Between two occurrences of state *II* in the path, the state *III* (surrounded with two boxes) appears twice. If we match every new occurrence of state *II* in the path with its first occurrence (we will call this way of matching the states *first state matching*), the path will be divided in 4 component paths:

$$\underbrace{Ia_0II}_{1}[\underbrace{\langle IIbIIIcIIIdII\rangle}_{2}\underbrace{\langle IIbIIIcIIIdII\rangle}_{3}]^{II}\underbrace{IIeIV}_{4}$$

If we do the same for *III* in the paths 2) *IIbIII cIIIdII* and 3) *IIbIIIcIIIdII* and eliminate all the states, we obtain the marked trace $a_0[\langle b[\langle c\rangle]^{III}d\rangle\langle b[\langle c\rangle]^{III}d\rangle]^{II}e$. This marked trace corresponds to the initial trace $a_0bcdbcde$. However, there are also other ways of transforming it into a marked trace. For example, the states of the same trace can be grouped in another way as $Ia_0IIb\boxed{III}c\boxed{III}dIIb\boxed{III}c\boxed{III}dIIeIV$ and the same trace has another correspondent marked trace which is $a_0b[\langle c\rangle\langle db\rangle\langle c\rangle]^{III}de$.

From this example we see that there is not a unique way of transforming a trace in a marked trace. We leave it as an option to the implementer (the user of our theory) to choose the way by which he transforms a trace into a marked trace. In the next subsection, we will give a particular way to implement the transformation of a trace in a marked trace and a way to obtain the set of marked traces which will be called the set of *representative marked traces*. If the implementer chooses another transformation, he can still use the theory presented in this paper if the correspondence between marked traces and traces is unique, and if its set of marked traces respects the property that the widths and the nesting depth are uniformly bounded. But we will come to this in the next subsection.

8.5.2 An algorithm for obtaining a set of representatives

In the beginning of this subsection we give a way to implement the transformation of a trace in a representative marked trace and to obtain the set of representatives.

This set is obtained by applying the following function (*Mark*) on each trace (path) of a finite-state minimal deterministic automaton s. The function builds a marked trace from a trace using a first state match technique like the one we used for the trace $a_0bcdbcde$ at the beginning of the previous example. In *Mark* we use the following function and procedure: 1) the function $RepetitiveState(p, Q)$, p a path, Q a set of states, returns true if there is a state of p which is contained in Q and which occurs more than once in p and 2) the procedure $Divide(p, Q, q, n, p_1, ..., p_n)$ finds $q \in Q$ and splits p in n parts $p_1, ..., p_n$ ($n \in \mathbf{N}, i = 2, ..., n-1$, p_1q, qp_iq, qp_n paths) such that: i) $q \in Q$ is the first repetitive state in p, ii) $p = p_1qp_2q...qp_n$ and iii) the set of states of p_j does not contain q ($j = 1, ..., n$). The operator $|_{trace}$ returns the trace corresponding to a path by eliminating all the states from the path.

```
function Mark (p : Path, Q : SetStates) : MarkedTrace;
```

```
var $q$ : State;
    $p_1, ..., p_n$ : $(\epsilon+Label)(State\ Label)^*(\epsilon+State)$;
begin
  if $(\neg RepetitiveState(p,\ Q))$ then
(1)     return $p\ |_{trace}$;
  else
(2)     $Divide(p,\ Q,\ q,\ n,\ p_1, ..., p_n)$;
(3)     $Q = Q \setminus \{q\}$;
(4)     return $Mark(p_1 q,\ Q)[\langle Mark(qp_2q,\ Q)\rangle ... \langle Mark(qp_{n-1}q,\ Q)\rangle]^q\ Mark(qp_n,\ Q)$;
end
```

Initially, the function *Mark* is applied to a path p and to the set of states Q of the automaton. When a) p does not contain states from Q which are repetitive, *Mark* returns the trace corresponding to p ($p\ |_{trace}$). When a) does not hold, *Mark* finds the first repetitive state $q \in Q$ in p and divides p in n parts $p_1, ..., p_n$. The first repetitive state q is the first state met when parsing the path p from left to right which has more than one occurrence in the path p. Every p_i ($i = 2, ..., n-1$) lies between two occurrences of q in p; p_1 and p_n are the initial part (followed by q) and the last part (preceded with q) of p, respectively. After this the state q is deleted from Q, which becomes $Q \setminus \{q\}$. In this way, terms as $[\_[\_]^q\_]^q$ in which q is interpreted twice as a cycle are avoided. Without deletion, the repetition of q could be reinterpreted as a cycle by a later call of *Mark*, when it is applied on a component path qp_iq. After the transformation of Q, *Mark* returns the concatenation of the marked traces obtained by recursively applying the algorithm to the components $p_1, ..., p_n$, which is $Mark(p_1q,\ Q)[\langle Mark(qp_2q,\ Q)\rangle ... \langle Mark(qp_{n-1}q,\ Q)\rangle]^q Mark(qp_n,\ Q)$.

Example Let us consider the path $p = Ia_0IIbIIIdIIeIV$ of the automaton from Figure 8.1 and the set of states of this automaton, $Q = \{I, II, III, IV\}$.
The call of $Mark(Ia_0IIbIIIdIIeIV, \{I, II, III, IV\})$ implies:

Apply (2) $Divide(Ia_0IIbIIIdIIeIV, \{I, II, III, IV\}, II, 3, Ia_0, bIIId, eIV)$
outs: $q = II$ and $p_1 = Ia_0$, $p_2 = bIIId$, $p_3 = eIV$

Apply (3) $Q = \{I, II, III, IV\} \setminus \{II\} = \{I, III, IV\}$

Apply (4) $Mark(Ia_0II, \{I, III, IV\})[\langle Mark(IIbIIIdII, \{I, III, IV\})\rangle]^{II} Mark(IIeIV, \{I, III, IV\})$
$\overset{\text{Apply (1)}}{=}\ Ia_0II\ |_{trace}\ [\langle IIbIIIdII\ |_{trace}\rangle]^{II} IIeIV\ |_{trace}$
$= a_0[\langle bd\rangle]^{II} e$

An implementation of this algorithm is given in the next chapter of the thesis. The set of representatives is $traces^m(s) = \{Mark(p, Q)\ |\ p \in path(s)\}$. In the remainder of this chapter, in the examples which we use, we assume that the marked traces are generated with *Mark* from the traces of an automaton.

As one can see, our way of building the set of representatives is rather complex. One can imagine easier solutions as for example: every marked trace is the trace itself. But the marked traces built with *Mark* have nice properties which are required for the application of our test selection theory. For example the width of such marked traces is uniformly bounded (Lemma 8.5.4), a property which is used in the theorem of total boundedness (Theorem 8.8.2). The marked traces generated with the trivial solution do not have this property.

Once we have the set of representatives, we want to know whether it has some specific properties.

For a marked trace of an automaton we want to know if the width of it is uniformly bounded, and if the nesting depth of it is also bounded. Here uniformly bounded means that the same upperbound applies at all nesting levels. But first let us define these terms.

In the remainder of this chapter, in the definitions which are given we use as the domain of different functions the set $\mathit{traces}^m(s)$ (for example in the definition below). For being strict formal, $\mathit{traces}^m(s)$ should be replaced with L_Q^* and it should be mentioned that the function defined is used only for the traces from $\mathit{traces}^m(s)$. This replacement is needed because it might be the case that if $ut \in \mathit{traces}^m(s)$ then $t \notin \mathit{traces}^m(s)$ (for example the trace correspondent to the marked trace t is not starting from the initial state; u is a label). For simplifying the presentation we will abuse notation by letting $\mathit{traces}^m(s)$ as the domain of the functions defined.

Definition 8.5.3 Let s be an automaton. Let L be the labelset and Q the set of states of s. Then the function $\mathit{width} : \mathit{traces}^m(s) \to \mathbf{N}$ is:

1. if t is a marked trace because t is ϵ then $\mathit{width}(t) = 0$;
2. if t is a marked trace because t is t' with t' a non-empty marked trace then $\mathit{width}(t) = \mathit{width}(t')$;
3. if t is a non-empty marked trace because t is ut' with $u \in L$ and t' a marked trace then $\mathit{width}(t) = 1 + \mathit{width}(t')$;
4. if t is a non-empty trace because t is $[s]^q t'$ with s a non-empty sequence, $q \in Q$ and t' a marked trace then $\mathit{width}(t) = 1 + \mathit{width}(t')$.

In the definition above the terms $[\_]^-$ are counted as single terms of the marked trace. Therefore it did not make sense to define the function *width* for non-empty sequences.

Example Let us take the trace $a_0[\langle bd\rangle\langle bd\rangle]^{II} f$. Then

$$\mathit{width}(a_0[\langle bd\rangle\langle bd\rangle]^{II} f) =$$

$$\mathit{width}(a_0) + \mathit{width}([\langle bd\rangle\langle bd\rangle]^{II}) + \mathit{width}(f) =$$

$$1 + 1 + 1 =$$

$$3$$

The following lemma shows that the width of every marked trace generated with *Mark* is uniformly bounded.

Lemma 8.5.4 *The width of a marked trace generated with Mark from an automaton and the widths of all its component marked traces are less than or equal to* $2m - 1$*, where m is the number of states of the automaton.*

For the proof see Appendix A.4.

Definition 8.5.5 Let s be an automaton. Let L be the labelset and Q the set of states of s. Then the function $\mathit{nesting} : \mathit{traces}^m(s) \to \mathbf{N}$ is:

1. if t is a marked trace because t is ϵ then $\mathit{nesting}(t) = 0$;

2. if t is a marked trace because t is t' with t' a non-empty marked trace then $\mathit{nesting}(t) = \mathit{nesting}(t')$;

3. if t is a non-empty marked trace because t is ut' with $u \in L$ and t' a marked trace then $\mathit{nesting}(t) = \mathit{nesting}(t')$;

4. if t is a non-empty marked trace because t is $[s]^q t'$ with s a non-empty sequence, $q \in Q$ and t' a marked trace then $\mathit{nesting}(t) = \mathit{max}(1 + \mathit{nesting}(s), \mathit{nesting}(t'))$;

5. if s is a non-empty sequence because s is $\langle t\rangle$ with t a non-empty marked trace then $\mathit{nesting}(s) = \mathit{nesting}(t)$;

6. if s is a non-empty sequence because s is $\langle t\rangle s'$ with t a non-empty marked trace and s' a non-empty sequence then $\mathit{nesting}(s) = \mathit{max}(\mathit{nesting}(t), \mathit{nesting}(s'))$.

Example Let us take the trace $a_0[\langle bd\rangle\langle b[\langle c\rangle]^{III}d\rangle]^{II}f$. Then

$$\mathit{nesting}(a_0[\langle bd\rangle\langle b[\langle c\rangle]^{III}d\rangle]^{II}f) =$$

$$\mathit{max}(1 + \mathit{nesting}(\langle bd\rangle\langle b[\langle c\rangle]^{III}d\rangle), \mathit{nesting}(f)) =$$

$$\mathit{max}(1 + \mathit{max}(\mathit{nesting}(bd), \mathit{nesting}(\langle b[\langle c\rangle]^{III}d\rangle)), 0) =$$

$$\mathit{max}(2, 0) =$$

$$2$$

The following lemma shows that the nesting depth of every marked trace generated with *Mark* is bounded.

Lemma 8.5.6 *The nesting depth of a marked trace generated with Mark from an automaton is less than or equal to the number of states of the automaton.*

For the proof see Appendix A.5.

As we motivated before, for applying our theory of test selection we need some specific properties for the set of representatives. So we require for the set of (representative) marked traces of an automaton that the width of every marked trace, the widths of all its component marked traces, and its nesting depth to be uniformly bounded. In this subsection we showed that the marked traces generated with *Mark* have these properties (Lemma 8.5.4, Lemma 8.5.6). Certain other algorithms work as well. For example, similarly as we did in this subsection, one can prove that the marked traces obtained with a *last state matching* technique (the last repetitive state of the path is matched) have also these properties. Independent of the way in which the set of marked traces is obtained, once it has the required properties, our test selection theory can be applied to it.

Now we have an algorithm that makes sure that every trace of the automaton has a unique correspondent representative marked trace, we will work with marked traces in place of traces throughout the remainder of this chapter.

8.6 The heuristics defined for marked traces

Below we will define the heuristics in a formal way. As we presented in Section 8.3.1, the intuition behind the heuristics *Reduction* and *Cycling* is that they take two aspects into account: the finiteness of 1) the number of outgoing transitions of certain states and of 2) the number of times each cycle can be traversed by every single trace.

When *Reduction* is applied, the labelset L is split in two parts: the selected labels which form a finite set $L' \subseteq L$ and the set of unselected labels which is $L \setminus L'$. This application can be seen as the application of a mapping function *trans*: $L \to L'$ which maps every unselected label to a selected label from L' and every selected label to itself. Without loss of generality let us assume that the labels label uniquely the transitions (otherwise *trans* should be defined on the product $Q \times L$, where Q is the set of states of the considered automaton). Then, for all $q \in Q$, if u labels a transition of state q then, *trans*(u) should also label a transition of state q, i.e. $\exists q' : q \overset{trans(u)}{\to} q'$. This property is needed for ensuring that *Reduction* produces valid traces of the automaton. One practical way to make the selection and to obtain L' and *trans* is by defining a distance d_L between labels, such that the metric space (L, d_L) is totally bounded. Let us fix a positive real number $\varepsilon_L \geq 0$. Now L' will be a labelset which is an ε_L-cover of L. A set of labels which are remote from each other (their distance is greater than ε_L) is selected and the labels from $L \setminus L'$ remain unselected. The function *trans*: $L \to L'$ can be defined in this case such that *trans*$(a) = b$ with $a \in L, b \in L'$ and $d_L(a, b)$ minimum.

For the *Cycling* heuristic we relate the cycles of the automaton to the marked representation of the trace; limiting the numbers of times of traversing the cycles means limiting the size of the non-empty sequences in a marked traces. Now, let us define these heuristics in a formal way.

Definition 8.6.1 Let s be an automaton. Let L be the labelset and Q the set of states of s. The labels label uniquely the transitions of the automaton s. Let $L' \subseteq L$ be a finite subset of L and let *trans*: $L \to L'$ be the mapping function such that if $u \in L$ labels a transition of state q then, *trans*(u) will also label a transition of state q. Then the heuristic *Reduction* : $traces^m(s) \to traces^m(s)$ is:

1. if t is a marked trace because t is ϵ then $Reduction(t) = \epsilon$;
2. if t is a marked trace because t is t' with t' a non-empty marked trace then $Reduction(t) = Reduction(t')$;
3. if t is a non-empty marked trace because t is ut' with $u \in L$ and t' a marked trace then $Reduction(t) = trans(u)Reduction(t')$;
4. if t is a non-empty trace because t is $[s]^q t'$ with s a non-empty sequence, $q \in Q$ and t' a marked trace then $Reduction(t) = [Reduction(s)]^q Reduction(t')$;
5. if s is a non-empty sequence because s is $\langle t \rangle$ with t a non-empty marked trace then $Reduction(s) = \langle Reduction(t) \rangle$;
6. if s is a non-empty sequence because s is $\langle t \rangle s'$ with t a non-empty marked trace and s' a non-empty sequence then $Reduction(s) = \langle Reduction(t) \rangle Reduction(s')$.

Example Let us consider the automaton from Figure 8.1. For this automaton the set of labels is $L = \{c, b, d, e, f\} \cup \{a_i \mid i = 0, 1, ...\}$.

Let $L' = \{a_0, c, b, d, e, f\}$, which is a finite subset of L and *trans*: $L \to L'$

$$trans(x) = \begin{cases} a_0 & x = a_i, i \in \mathbf{N} \\ x & \text{otherwise} \end{cases}$$

Then

$Reduction(a_3e) =$

$Reduction(a_3)Reduction(e) =$

$trans(a_3)trans(e) =$

a_0e

Let s be a non-empty sequence. By $\mid s \mid$ we mean the size of the non-empty sequence, i.e. the numbers of terms of type $\langle \_ \rangle$ in the sequence s.

Definition 8.6.2 The size of a non-empty sequence is

1. if s is a non-empty sequence because s is $\langle t \rangle$ with t a non-empty marked trace then $\mid s \mid = 1$;
2. if s is a non-empty sequence because s is $\langle t \rangle s'$ with t a non-empty marked trace and s' a non-empty sequence then $\mid s \mid = 1 + \mid s' \mid$

The definition for the *Cycling* heuristic is given below.

Definition 8.6.3 Let s be an automaton. Let L be the labelset and Q the set of states of s. Let l_c be the cycle limit with $l_c \in \mathbf{N}$. Then the heuristic $Cycling : traces^m(s) \rightarrow traces^m(s)$ is:

1. if t is a marked trace because t is ϵ then $Cycling(t) = \epsilon$;
2. if t is a marked trace because t is t' with t' a non-empty marked trace then $Cycling(t) = Cycling(t')$;
3. if t is a non-empty marked trace because t is ut' with $u \in L$ and t' a marked trace then $Cycling(t) = uCycling(t')$;
4. if t is a non-empty trace because t is $[s]^q t'$ with s a non-empty sequence, $q \in Q$ and t' a marked trace then
   (a) $Cycling(t) = Cycling(t')$ if $l_c = 0$;
   (b) $Cycling(t) = [Cycling(s)]^q Cycling(t')$ if $l_c > 0$;
5. if s is a non-empty sequence because s is $\langle t \rangle$ with t a non-empty marked trace then $Cycling(s) = \langle Cycling(t) \rangle$;
6. if s is a non-empty sequence because s is $\langle t \rangle s'$ with t a non-empty marked trace and s' a non-empty sequence then
   (a) $Cycling(s) = \langle Cycling(t) \rangle$ if $l_c = 1$;
   (b) $Cycling(s) = \langle Cycling(t) \rangle Cycling(s')$ if $l_c \geq \mid s \mid$;
   (c) $Cycling(s) = \langle Cycling(t) \rangle Cycling(s'')$ if $l_c < \mid s \mid$; s'' is obtained by eliminating the last terms (of type $\langle \_ \rangle$) from s' after $l_c - 1$ positions (if $s' = \langle t_1 \rangle .... \langle t_{n-1} \rangle \langle t_n \rangle$ then $s'' = \langle t_1 \rangle .... \langle t_{l_c - 1} \rangle$ with $n = \mid s' \mid$, $i \leq n$ and t_i non-empty marked traces).

In the definition above, for a non-empty sequence s of type $\langle t\rangle s'$, by cutting the last terms of s' when the size of the sequence $\langle t\rangle s'$ (or s) exceeds l_c will preserve the first l_c terms of s.

One observation related to the *Cycling* heuristic is that, in the current form, *Cycling* allows the repetition of the same cycle a number of l_c times. One option for improvement is to allow only cycles which are different (for the theory presented here it was too complicated to consider this).

Example Let us consider the automaton from Figure 8.1. Let us fix l_c to 2. Then

$$Cycling(a_0[\langle bd\rangle\langle bd\rangle\langle bd\rangle]^{II}e) =$$

$$Cycling(a_0)Cycling([\langle bd\rangle\langle bd\rangle\langle bd\rangle]^{II})Cycling(e) =$$

$$a_0[\langle bd\rangle\langle bd\rangle]^{II}e$$

Lemma 8.6.4 *Reduction(Cycling(x)) = Cycling(Reduction(x))*

For the proof see Appendix A.6.

8.7 A trace distance for marked traces

In this section we make the trace distance more precise, defining it as a distance function according to Definition 8.4.2. As explained in Section 8.4, this gives us an alternative formalization of the ideas behind the heuristics (they will be compared in Section 8.8). We will combine these ideas with another well-known idea, viz. the edit distance. Section 8.7.1 introduces the edit distance. After this preparation, the definition of the trace distance function can be given (Section 8.7.2).

8.7.1 The edit distance between strings

Because in our trace distance we use the concept of edit distance we shall present this first. The concept is applied in problems such as string search, words substitution using dictionaries, etc. Informally the edit distance is defined as the minimum number of insertions, deletions and substitutions required to transform one string into another.

Levenshtein ([Ste92]) defined the edit distance $d(x, y)$ between two strings x and y as the minimum of the cost of editing x to transform it into y. The cost of editing is the sum of the costs of a number of atomic edit actions. According to Levenshtein the costs are as follows: inserting a symbol costs 1, deleting a symbol costs 1 and changing a symbol into another symbol costs 1 too.

Wagner and Fisher ([Ste92]) generalized the definition of Levenshtein by adopting different costs (weights) for the various atomic edit actions. According to Wagner-Fisher transforming a into a b costs $w(a, b)$. Extending this notation, $w(a, \epsilon)$ is the cost of deleting a and $w(\epsilon, b)$ is the cost of inserting b. Again, the cost of editing is the sum of the costs of the atomic edit actions, and $d(x, y)$ is the minimum cost over all possible edit sequences that transform x into y.

Definition 8.7.1 Let $w(a, b)$ be the cost of transforming symbol a into symbol b, $w(a, \epsilon)$ be the cost of deleting a and $w(\epsilon, b)$ be the cost of inserting b. Of course $w(a, a) = 0$. Then the edit distance between the strings x and y is denoted as $ED(x, y)$ and it is defined by:

1. $ED(\epsilon, \epsilon) = 0$;
2. $ED(\epsilon, bv) = w(\epsilon, b) + ED(\epsilon, v)$;
   $ED(au, \epsilon) = w(a, \epsilon) + ED(u, \epsilon)$;

3. $ED(au, bv) = min(w(a, b) + ED(u, v), w(a, \epsilon) + ED(u, bv), w(\epsilon, b) + ED(au, v))$.

where a, b are symbols and u, v are strings.

This definition will be used throughout the chapter.

Example Let us take the labelset $L = \{a, b, c\}$ with the cost 1 for insertion, deletion, and for transforming a symbol in another symbol. The edit distance between a and ba is computed as:

$$ED(a, ba) =$$

$$min(w(a, b) + ED(\epsilon, a), w(a, \epsilon) + ED(\epsilon, ba), w(\epsilon, b) + ED(a, a)) =$$

$$min(1 + w(\epsilon, a) + ED(\epsilon, \epsilon), 1 + w(\epsilon, b) + w(\epsilon, a) + ED(\epsilon, \epsilon),$$

$$1 + min(w(a, a) + ED(\epsilon, \epsilon), w(a, \epsilon) + ED(\epsilon, a), w(\epsilon, a) + ED(a, \epsilon))) =$$

$$min(1 + 1, 1 + 2, 1 + 0) =$$

$$1$$

So the edit distance between a and ba is 1 which corresponds to the insertion of b.

8.7.2 Defining a trace distance

Our test selection technique uses two heuristics. For expressing these heuristics in the trace distance, it is important to remember that in the formalization of the *Reduction* heuristic a label distance was used. The incorporation of this heuristic in the trace distance is achieved in a simple way by using the label distance in the formula of the trace distance. Now a solution should be found for the *Cycling* heuristic.

For the *Cycling* heuristic we simply weight every level k (by level we mean the position of a term $\langle\_\rangle$) of a cycling symbol (a marked trace of type $[\_]^q$, $q \in Q$) with a weight from a descending series of positive numbers p_k (p_k can take also the value 0). This series has the property that $\sum_{k=1}^{\infty} p_k = 1$. The logic behind this weighting is that summing the weights after a given limit (which is the cycle limit) will contribute with a small number. This reflects our assumption that the first cycles are more important than the later cycles. Moreover the first cycles will have assigned larger weights than the later cycles because the series is descending.

We will define the trace distance for all the possible combinations of the grammar of marked traces, i.e. Definition 8.5.1, (which is generating marked traces). It is easy to observe that the distance for the marked traces is a straightforward extension of the distance for the non-empty marked traces. Therefore we will concentrate on the distance for the non-empty marked traces. We summarize the combinations below

- between the non-empty marked traces generated by labels (such as $a_0 \in L$) we will define a distance function called *AtomicDistance* because these are the atomic elements which form the marked trace; of course the *AtomicDistance* between two labels will be given by d_L, the distance between these labels;
- between the non-empty marked traces generated by cycles (such as building $[\langle bd\rangle\langle bd\rangle]^{II}$ once we know that $\langle bd\rangle\langle bd\rangle$ is a non-empty sequence) we employ the principle that cycles of different marks are very remote and hence have the maximum distance, i.e, 1; when dealing with

cycles of the same mark we employ weighting factors p_k with the effect that the later iterations are considered less important than e.g. the first iteration; this can be done by using a function *EditDistanceWeighted* which is an edit distance for which the formula of Definition 8.7.1 is modified in such a way to take the weights into account;

- between the non-empty marked traces generated by concatenation (such as $a_0 f$ or $[\langle bd\rangle]^{II}e$) we will use a distance function called *EditDistance*; we took this option because these traces are generated in a similar style as the strings are formed and it is quite natural to use it because it compares the terms which form the marked traces in a good way (for example in the traces a_0e and $a_0[\langle bd\rangle]^{II}e$ the edit distance will recognize that the labels a_0 and e from the first trace are present in the second trace).

The rest of the possible combinations are defined in a similar style by using one of the techniques mentioned above (*EditDistance* or *AtomicDistance*).

We observe also that this trace distance is to be used in the computation of a coverage which should be in the range [0, 1]. For simplifying the computation of coverage, we want the trace distance values to be in the range [0, 1]. This can be done by dividing all the above mentioned values (generated with an *EditDistance* or *AtomicDistance*) by the maximum width of the marked traces from $traces^m(s)$ (the maximum width is finite, see Section 8.5.2). For completing the picture it is necessary to add that the trace distance between a null trace (ϵ) and any other marked trace is maximum (1).

Now we have all the ingredients to define a trace distance on marked traces. We will call it d. In the definition, the distances already mentioned (*EditDistance* and *AtomicDistance*) will be used; also it is implicitly assumed that the definition is symmetric in the sense that $d(x, y) = d(y, x)$, x and y being marked traces and that $d(x, x) = 0$.

As explained above (first bullet), the function *AtomicDistance* deals with the cases, $a \in L$ and $[\_]^-$. We generalize it to marked traces of the form $[\_]^-$ as well.

Definition 8.7.2 Let s be an automaton. Let L be the labelset of s, d_L the label distance defined on it and Q the set of states of s. Assume that the metric space (L, d_L) is totally bounded and d_L has all its values in the range [0, 1]. Let l_m be the maximum of the width of the marked traces from $traces^m(s)$. Let p_k $(k = 1, 2, ...)$ be a descending series of positive numbers such that $\sum_{k=1}^{\infty} p_k = 1$. The trace distance d is symmetric in the sense that $d(x, y) = d(y, x)$, x and y being marked traces and that $d(x, x) = 0$. The same properties hold for the auxiliary distances *EditDistance*, *AtomicDistance* and *EditDistanceWeighted*. Then the distance $d : traces^m(s) \times traces^m(s) \to [0, 1]$ is defined by:

1. (a) $d(t, t') = \frac{EditDistance(t,t')}{l_m}$;

   (b) $d(t, \epsilon) = 1$;

   (c) $d(\epsilon, \epsilon) = 0$;

   with t, t' non-empty marked traces;

2. (a) $EditDistance(\epsilon, u') = 1$;

   (b) $EditDistance(\epsilon, u'v') = EditDistance(\epsilon, u') + EditDistance(\epsilon, v')$;

   (c) $EditDistance(u, u') = AtomicDistance(u, u')$;

   (d) $EditDistance(u, u'v') = min(EditDistance(u, u') + EditDistance(\epsilon, v'),$
   $EditDistance(u, \epsilon) + EditDistance(\epsilon, u'v'),$
   $EditDistance(\epsilon, u') + EditDistance(u, v'))$;

(e) $EditDistance(uv, u'v') = min(EditDistance(u, u') + EditDistance(v, v'),$
$EditDistance(u, \epsilon) + EditDistance(v, u'v'),$
$EditDistance(\epsilon, u') + EditDistance(uv, v'));$

with u ($u \in L$ or $u = [\_]^{\text{-}}$), u' ($u' \in L$ or $u' = [\_]^{\text{-}}$), v, v' non-empty marked traces;

3. $$AtomicDistance(x, x') = \begin{cases} d_L(x, x') & x, x' \in L \\ EditDistanceWeighted^1(s, s') & x = [s]^q, x' = [s']^q, q \in Q \\ 1 & \text{otherwise} \end{cases}$$

with $x \in L$ or $x = [\_]^{\text{-}}$ and $y \in L$ or $y = [\_]^{\text{-}}$;

4. (a) $EditDistanceWeighted^k(\epsilon, \langle t' \rangle) = p_k \times d(\epsilon, t');$

(b) $EditDistanceWeighted^k(\epsilon, \langle t' \rangle s') = p_k \times d(\epsilon, t') + EditDistanceWeighted^{k+1}(\epsilon, s');$

(c) $EditDistanceWeighted^k(\langle t \rangle, \langle t' \rangle) = p_k \times d(t, t');$

(d) $EditDistanceWeighted^k(\langle t \rangle, \langle t' \rangle s') = min(p_k \times d(t, t') + EditDistanceWeighted^{k+1}(\epsilon, s'),$
$p_k \times d(t, \epsilon) + EditDistanceWeighted^{k+1}(\epsilon, \langle t' \rangle s'),$
$p_k \times d(\epsilon, t') + EditDistanceWeighted^{k+1}(\langle t \rangle, s'));$

(e) $EditDistanceWeighted^k(\langle t \rangle s, \langle t' \rangle s') = min(p_k \times d(t, t') + EditDistanceWeighted^{k+1}(s, s'),$
$p_k \times d(t, \epsilon) + EditDistanceWeighted^{k+1}(s, \langle t' \rangle s'),$
$p_k \times d(\epsilon, t') + EditDistanceWeighted^{k+1}(\langle t \rangle s, s'));$

with t, t' non-empty marked traces, s, s' non-empty sequences.

We add some explanation. For determining the maximum of the width of the marked traces l_m extra computational effort is needed (l_m can be computed based on the particular form of the automaton considered or it can be approximated). It is easy to check that the definition of *EditDistance* and *EditDistanceWeighted* are based on Definition 8.7.1 except for the fact that suitable weighting factors p_k have been incorporated and their formulas have been extended for more basic cases. In the formula of *EditDistance*, at the point 2c of the definition, it is also easy to check that the call of *AtomicDistance* which corresponds to the transformation of u in u' gives the minimum of all possible edit operations (a deletion and an insertion cost 1). Therefore at that point we gave directly the formula which gives the minimum. In a similar style *EditDistanceWeighted* will be defined. The metric space $(traces^m(s), d)$ is indeed a metric space because it is built on the metric space (L, d_L) with operations of positive weighting, summations and edit distances, operations which preserve the properties of metric spaces. The complete proof is given in Lemma 8.7.3.

The parameter k in $EditDistanceWeighted^k$ indicates the position at which the next edit action takes place. Please note that the recursive definition of $EditDistanceWeighted^k$ is well defined because at least one of the right-hand sides of the equation is one symbol shorter than the corresponding left-hand side (therefore, the fact that k is increasing causes no problem). The base cases are $EditDistanceWeighted^k(\epsilon, \langle t' \rangle)$ and $EditDistanceWeighted^k(\langle t \rangle, \langle t' \rangle)$. We will illustrate how the *EditDistanceWeighted* function works. Let us consider two strings ab and c, (although *EditDistanceWeighted* is defined on marked traces, for making the example more understandable, let us show how it works on common strings). For transforming one string into another, there are five possible combinations: 1)$(ab, c\epsilon)$, 2)$(ab, \epsilon c)$, 3)$(\epsilon ab, c\epsilon\epsilon)$, 4)$(a\epsilon b, \epsilon c\epsilon)$, 5)$(ab\epsilon, \epsilon\epsilon c)$. The *EditDistanceWeighted* will be the minimum from the costs of the five combinations. The cost for every combination is computed as the sum of the edit actions multiplied by the corresponding weights. For example, in the combination $(ab\epsilon, \epsilon\epsilon c)$, which means two deletions followed by one insertion, we encounter weighting factors for $d(a, \varepsilon)$, $d(b, \varepsilon)$ and $d(\epsilon, c)$, which are p_1, p_2 and p_3, respectively. The cost of this combination will be $p_1 \times d(a, \varepsilon) + p_2 \times d(b, \varepsilon) + p_3 \times d(\varepsilon, c)$. In a similar style the cost of the other combinations is computed and the minimum is chosen for *EditDistanceWeighted*.

Example Let us consider the automaton from Figure 8.1. For this automaton the maximum width of the marked trace is 3. Let $p_k = \frac{1}{2^k}, k = 1, 2, ...$ Let d_L be the following label distance

$$d_L(x, y) = \begin{cases} 0 & x = y \\ \mid \frac{1}{4^i+1} - \frac{1}{4^j+1} \mid & x = a_i, y = a_j, i, j \in \mathbf{N} \\ 1 & \text{otherwise} \end{cases}$$

1. Let us consider the traces a_0e and $a_0[\langle bd \rangle]^{II}e$.

   $$d(a_0e, a_0[\langle bd \rangle]^{II}e) =$$
   $$\frac{1}{3} \times (\mathit{EditDistance}(a_0e, a_0[\langle bd \rangle]^{II}e)) =$$
   $$\frac{1}{3} \times (\mathit{EditDistance}(\epsilon, [\langle bd \rangle]^{II})) =$$
   $$\frac{1}{3}$$

   The trace distance recognizes that the symbols a_0 and e from the first trace are present in the second trace.

2. *The cycling effect:*

   Let us take the traces $a_0[\langle bd \rangle]^{II}e$, $a_0[\langle bd \rangle\langle bd \rangle]^{II}e$ and $a_0[\langle bd \rangle\langle bd \rangle\langle bd \rangle]^{II}e$;

   $$d(a_0[\langle bd \rangle]^{II}e, a_0[\langle bd \rangle\langle bd \rangle]^{II}e) =$$
   $$\frac{1}{3} \times \mathit{EditDistance}(a_0[\langle bd \rangle]^{II}e, a_0[\langle bd \rangle\langle bd \rangle]^{II}e) =$$
   $$\frac{1}{3} \times \mathit{AtomicDistance}([\langle bd \rangle]^{II}, [\langle bd \rangle\langle bd \rangle]^{II}) =$$
   $$\frac{1}{3} \times \mathit{EditDistanceWeighted}^1(\langle bd \rangle, \langle bd \rangle\langle bd \rangle) =$$
   $$\frac{p_2}{3} =$$
   $$\frac{1}{12}$$
   $$d(a_0[\langle bd \rangle\langle bd \rangle]^{II}e, a_0[\langle bd \rangle\langle bd \rangle\langle bd \rangle]^{II}e) =$$
   $$\frac{1}{3} \times \mathit{EditDistance}(a_0[\langle bd \rangle\langle bd \rangle]^{II}e, a_0[\langle bd \rangle\langle bd \rangle\langle bd \rangle]^{II}e) =$$
   $$\frac{1}{3} \times \mathit{AtomicDistance}([\langle bd \rangle\langle bd \rangle]^{II}, [\langle bd \rangle\langle bd \rangle\langle bd \rangle]^{II}) =$$
   $$\frac{1}{3} \times \mathit{EditDistanceWeighted}^1(\langle bd \rangle\langle bd \rangle, \langle bd \rangle\langle bd \rangle\langle bd \rangle) =$$

$$\frac{p_3}{3} =$$
$$\frac{1}{24}$$

When two marked traces are cycling more times through the same cycle, the values of the trace distance start to decrease.

3. *The reduction effect:*

   Let us take the traces a_0e, a_1e and a_2e

$$d(a_0e, a_2e) = \frac{1}{3} \times (EditDistance(a_0e, a_2e)) =$$
$$\frac{d_L(a_0, a_2)}{3} =$$
$$\frac{\frac{15}{64}}{3} =$$
$$\frac{5}{64}$$
$$d(a_1e, a_2e) = \frac{1}{3} \times (EditDistance(a_1e, a_2e)) =$$
$$\frac{d_L(a_1, a_2)}{3} =$$
$$\frac{\frac{3}{64}}{3} =$$
$$\frac{1}{64}$$

   When the label distance between the labels (which compose the marked traces) is decreasing, the trace distance is also decreasing.

Lemma 8.7.3 *Let s be an automaton.*

1. *All the values of d are in the range $[0, 1]$, i.e. $\forall x, y \in traces^m(s) : d(x, y) \leq 1$.*
2. *The space $(traces^m(s), d)$ is a metric space.*

For the proof see Appendix A.7.

8.8 Transforming the heuristics into a coverage

The trace distance formula depends on the label distance d_L which implements the *Reduction* heuristic and the weights p_k which implement the *Cycling* heuristic. On the other hand, by choosing for each automaton s a finite set $L_\varepsilon \subseteq L$ which is an ε_L-cover of L with respect to d_L and a cycling limit l_c, a finite set of marked traces $T \subseteq traces^m(s)$ can be obtained. This is done by the application of the *Cycling* and the *Reduction* heuristics on $traces^m(s)$ by taking $T = Ran(Cycling \circ Reduction)$. Now for this T and using d we want to compute the ε value such that T is an ε-cover of $traces^m(s)$ so that we can compute the coverage $cov(T, traces^m(s))$ – see Definition 8.4.6. Intuitively ε should depend on ε_L and l_c. Its formula is given by Theorem 8.8.1. In the remainder of this chapter we assume that $l_c \geq 1$.

The next theorems are about other properties of the trace distance d: Theorem 8.8.2 shows that for any desired ε it is possible to obtain an ε-cover by choosing a suitable cycle limit and a suitable label approximation, with other words that the metric space $(traces^m(s), d)$ is a metric space that is totally bounded. Theorem 8.8.3 shows that the distance d implements the *Reduction* and *Cycling* heuristics.

For the following theorems, we assume that all the ingredients for defining the metric space $(traces^m(s), d)$ are given. We will enumerate them below. Let s be an automaton. Let L be the labelset of s and d_L the label distance defined on it. Assume that the metric space (L, d_L) is totally bounded and d_L has all its values in the range [0, 1]. Let p_k ($k = 1, 2, ...$) be a descending series of positive numbers such that $\sum_{k=1}^{\infty} p_k = 1$. Let l_m be the maximum of the width from traces$^m(s)$. These are the ingredients needed for a proper definition of the metric space $(traces^m(s), d)$. Moreover, for the following theorems we consider that z, the maximum of the nesting depth of the marked traces, is also known.

Theorem 8.8.1 *Let* $(\text{traces}^m(s), d)$ *be a metric space. Let l_c be the cycle limit. Let $L_\varepsilon \subseteq L$ be an ε_L-cover of L. Let $\varepsilon \in$* [0, 1] *be a positive number computed in the following way:*

1. $\varepsilon = \varepsilon^z$ *with*

(a) $\varepsilon^0 = \varepsilon_L$*;*

(b) for $i = 1, ..., z$ *:*

i. $\varepsilon_c^i = \sum_{k=1}^{l_c} p_k \times (\max_{j=0,...,i-1}(\varepsilon^j)) + \sum_{k=l_c+1}^{\infty} p_k$*;*

ii. $\varepsilon^i = \max_{\text{cycles}=0,...,l_m}(\frac{\text{cycles} \times \varepsilon_c^i + (l_m - \text{cycles}) \times \varepsilon_L}{l_m})$.

Then the finite set T = Ran(Reduction ∘ Cycling) of traces obtained by the application of the two heuristics on traces$^m(s)$ is an ε-cover of traces$^m(s)$.

For the proof see Appendix A.8.

The following theorem shows that the metric space $(traces^m(s), d)$ is a totally bounded metric space.

Theorem 8.8.2 *Let* $(\text{traces}^m(s), d)$ *be a metric space. Then for every $\varepsilon \in$* [0, 1], *there exists a cycling limit l_c and a label approximation ε_L with $\varepsilon_L = \sum_{k=l_c+1}^{\infty} p_k \leq \frac{\varepsilon}{2^z}$ such that the finite set T = Ran(Reduction ∘ Cycling) of traces obtained by the application of the two heuristics on* traces^m *is an ε-cover of traces$^m(s)$ and the metric space $(traces^m(s), d)$ is totally bounded.*

For the proof see Appendix A.9.

The following theorem shows that the trace distance d implements the *Cycling* and the *Reduction* heuristics, in the sense of Definition 8.4.5.

Theorem 8.8.3 *Let* $(\text{traces}^m(s), d)$ *be a metric space. Let l_c be the cycle limit. Then the distance d implements the Reduction and the Cycling heuristics.*

For the proof see Appendix A.10.

For the computation of the coverage we approximate the minimum ε_m from Definition 8.4.6 with the ε^z computed in Theorem 8.8.1. We will illustrate the computation of the coverage in the following example.

Example Consider the automaton from Figure 8.1. Let us fix the final state to be *IV*. Let us consider the reduced set $L_\varepsilon = \{a_0, b, c, d, e, f\}$ which is an ε_L-cover of the labelset L with $\varepsilon_L = 0.25$ (ε_L is computed with respect to the d_L defined in the example from Section 8.7.2). For this automaton the maximum width is $l_m = 3$ and the maximum nesting depth is $z = 2$. Let us fix the series $p_k = \frac{1}{2^k}$ ($k \in \mathbf{N}$) and in the beginning $l_c = 1$.

Then the set of traces T which is obtained by the application of the heuristics *Reduction* and *Cycling* is an ε-cover of the whole set of traces $traces^m(s)$ with ε computed with the formula from Theorem 8.8.1 as:

1. $l_c = 1$, $L_\varepsilon = \{a_0, b, c, d, e, f\}$
   $\varepsilon^0 = \varepsilon_L = 0.25$;
   $\varepsilon_c^1 = \sum_{k=1}^{1} p_k \times \varepsilon^0 + \sum_{k=l_c+1}^{\infty} p_k = \frac{0.25}{2} + \sum_{k=l_c+1}^{\infty} \frac{1}{2^k} = 0.63$;
   $\varepsilon^1 = max_{\text{cycles}=0,\dots,3}(\frac{\text{cycles}\times\varepsilon_c^1 + (l_m - \text{cycles})\times\varepsilon_L}{l_m}) = 0.63$;
   $\varepsilon = \varepsilon^2 = 0.81$;

   The coverage is computed via Definition 8.4.6 and we use the epsilon approximation of Theorem 8.8.1. The coverage is $cov(T, traces^m(s)) \approx 1 - \varepsilon \approx 0.19$;

2. $l_c = 2$, $L_\varepsilon = \{a_0, b, c, d, e, f\}$

   When we enlarge the set T to T' for $l_c = 2$ we find that $cov(T', traces^m(s)) \approx 0.51$;

3. $l_c = 1$, $L_{\varepsilon'} = \{a_0, a_1, b, c, d, e, f\}$

   When we enlarge the set T to T'' for $L_{\varepsilon''} = \{a_0, a_1, b, c, d, e, f\}$ we find that $\varepsilon''_L = 0.06$ and that $cov(T'', traces^m(s)) \approx 0.29$.

In this example, one can see that the coverage increases more by adopting a higher value for the cycling limit than by using a larger label subset. Consequently, one might conclude that it is better to increase the cycling limit for obtaining a better coverage. But this is not always true because we defined specific values for p_k and d_L (for other values, to increase the label subset will be better).

It can be seen from this example that the monotonicity property required in [BTV91] that $T \subseteq T' \Rightarrow cov(T) \leq cov(T')$ is respected by our coverage (the coverage is approximated using the epsilon formula from Theorem 8.8.1). From an intuitive point of view this property is reasonable: if one generates more tests, then the coverage increases.

We want to prove it in the general case. For this we will make an assumption which is quite natural that a monotonicity property holds also for the metric space (L, d_L): for a label set L when we have L_ε and $L_{\varepsilon'}$ such that these sets are an ε_L-cover and respectively an ε'_L-cover of L, $L_\varepsilon \subseteq L_{\varepsilon'}$ then $\varepsilon_L \geq \varepsilon'_L$. The following theorem holds when the coverage is approximated using the epsilon formula from Theorem 8.8.1.

Theorem 8.8.4 *Let* (traces$^m(s), d$) *be a metric space. Let l_c and l'_c be two cycle limits ($l_c \leq l'_c$). Let $L_\varepsilon \subseteq L$ be an ε_L-cover of L and $L_{\varepsilon'} \subseteq L$ be an ε'_L-cover of L such that $L_\varepsilon \subseteq L_{\varepsilon'}$ and $\varepsilon_L \geq \varepsilon'_L$. Let T = Ran(Reduction $\circ$ Cycling) and T' = Ran(Reduction $\circ$ Cycling) be the two finite sets of traces obtained by the application of the two heuristics on $traces^m(s)$ using L_ε, l_c and respectively $L_{\varepsilon'}$, l'_c. Then $cov(T, traces^m(s)) \leq cov(T', traces^m(s))$.*

For the proof see Appendix A.11.

8.9 Conclusions

In this chapter we have studied two heuristics for reducing the number of traces in a test suite. The underlying assumption is that when automatically generating a set of traces, many traces will show similar behaviour. That means that deleting some traces will only reduce the error detection power slightly.

The first heuristic studied deals with restricting the branching degree of the nodes, when representing a set of test cases as a finite automaton. The basic idea is that in practice a high branching degree is generated because at the branching point an action is allowed which is parameterized by an element from a (large) data domain. The observation is that only a few values from such a data domain will show essentially different behaviour.

The second heuristic concerns the number of times a cycle in a finite automaton representation may be traversed. This is connected to the assumption that only for a few numbers of traversals the test cases will show essentially different behaviour.

The fact that we studied only these two heuristics in this chapter, does not mean that these are the only interesting heuristics. More heuristics can be defined, e.g. with respect to the general length of a trace and with respect to the uniformity of the number of outgoing transitions from a state. We embedded the two heuristics chosen in a more general framework which allows the extension of our work with other heuristics.

A heuristic is a general guideline for reducing test suites, which must be made more precise to be practically applicable. Especially for the cycling heuristic we had to introduce additional notation. The reason is that the cycling structure of a trace through a finite automaton must be made explicit. We introduced *marked traces* for this purpose, which enabled us to extend the work on cycle reduction by Vuong [ACV93, ACV97].

In order to introduce a notion of *coverage* for the test suites reduced by means of the above mentioned heuristics, we defined a *trace distance* on marked traces. The results of our studies can be used to effectively calculate the coverage of a test suite reduced with our techniques.

Although our formal definitions of *Reduction* and *Cycling* work on large, sometimes even infinite sets this need not cause practical or algorithmic problems. For example, the practical generation of traces could be done in a similar way to [ACV97], starting with a first trace and then suppress the generation of a subsequent trace if it is close to an already generated trace. Other solutions could be based on a suitable transformation of the automaton, as in fact we did in Figure 8.1. A similar remark applies to the calculation of the ε value for a generated test suite. A solution is to choose l_c and ε_L and calculate ε arithmetically using the results of Section 8.8.

One issue deserves attention, viz. the choice of representatives embodied in the algorithms of Sections 8.5.1 and 8.5.2. At first sight, the reader may think that the distance d defined in Section 8.7.2 is independent of the choice of the representatives. However, this does not hold in general because of the essential ambiguity in the concept of *cycle* when using an automaton as a specification.

The proposed test selection technique can be compared to the existing theories in this area. In particular, these are the hypothesis theory developed by [CG97] and the trace distance theory of [ACV93, ACV97]. The hypothesis theory embodies the trace distance theory (see [CG97]), but the nice thing about trace distance theory is that it gives a measure for the degree to which a reduced set of traces approximates the original one. So we chose an approach which combines these two theories. In our view, first the heuristics (test hypotheses in the theory of [CG97]) are to be defined. After that, based on these heuristics a trace distance is built. This gives the possibility to make a test selection with a given ε approximation. The change of the heuristics leads to the change of the trace distance used in test selection.

Another restricting requirement is that we assume the specification to be given as a minimal finite deterministic automaton. Some test generation tools already provide such a format, but others support general finite automata. Determinizing a finite automaton may cost exponential time. In this case it would be interesting to know whether the theoretical results achieved in this chapter could be extended to non deterministic automata.

An implementation of the theory presented in this Chapter is given in the next chapter of the thesis.

Chapter 9

Implementing the test selection theory

9.1 Introduction

In chapter 8 we elaborated an abstract theory for test selection. The link between the theory and the practice was not in the focus of the previous chapter. The current chapter fills this gap by presenting ways of implementing the abstract formulas of test selection. Several examples of useful distances are given. These examples add plausibility to the claim that the theory is useful in practice. Moreover they are a source of inspiration for other testers which want to use the theory of test selection presented in this thesis. Also the work described in this chapter is a step forward in using this selection theory in the TorX testing environment.

The implementation of the abstract theory of test selection is in the form of a C program. The C program is not connected to TorX. It is an independent module. The functions contained in this program form the basis of a new module which can be linked to TorX later. While describing the main features of this program, different ways of connecting it to TorX will be presented. Moreover we will indicate other possible connections to other test generation tools like Autolink. This is one of the reasons for which we kept the module tool-independent.

We tried to keep the program simple to make its understanding easy, but general enough to be used in realistic application domains such as classical public telephony and to be connected later to TorX or other test generation tools. Keeping the program simple was possible by making it tool independent. In this way it does not inherit complicated data structures and algorithms that are specific for e.g. TorX.

In this chapter we will not present all the functions of the C program in detail. The full program can be found at http://www.win.tue.nl/~goga/TestSelection.c. In this chapter, only its main algorithms will be presented. The program contains two modules which we will call *Unfold* and *Distance*.

- The module *Unfold* implements mainly the *Cycling* heuristic.
- The module *Distance* contains functions for computing distances. Because the *Reduction* heuristic is based on a label distance, we do not need a separate module for it. This heuristic will be explained when presenting the *Distance* module.

In this chapter, we will describe also a specification of a telephone. This specification was built together with researchers from KPN, one of the Dutch telephony operators, within CdR. Using the phone specification, we will illustrate the execution of different functions of the C program.

In Section 9.2 we present possible ways of connecting the C program to TorX and other test generation tools. We will concentrate on the module *Unfold*. In Section 9.3 we explain the data types of the program. The data types are used by both modules. The phone specification is described in Section 9.4. In Section 9.5, the main algorithms of *Unfold*, which implement the *Cycling* heuristic are given. Because the *Reduction* heuristic is implemented by using a label distance, we will describe it while presenting the label distance in Section 9.6. In the same section other algorithms for distance computation are given. Section 9.7 outlines the conclusions.

9.2 Connecting the C program to test generation tools

Before explaining the data structures of the program, we will discuss the main function of the *Unfold* module and how this module can be connected to TorX and other test derivation tools, such as Autolink. This will give some intuition regarding the design of the program. The main function of the *Unfold* module is called *Unfold* like the name of the module itself. This function is described in Section 9.5. Let us consider an automaton of a specification. The objective of the *Unfold* function is to generate all the traces which do not cycle more than a cycle limit, l_c, through the states of the automaton. The set of traces generated by *Unfold* can be seen as the test purpose (automatically generated) which is used by TorX to derive test cases. For TorX to work with test purposes, the *ioco* test derivation algorithm from Chapter 2 which is implemented by TorX is modified slightly such that for each trace from the set a test case is generated. A module which will be integrated in TorX and which will work with test purposes is under development at UT (University of Twente).

As we explained, the set of traces derived with *Unfold* can represent a test purpose objective for TorX. But not only for this tool such a set of traces can represent a test purpose. Let us take another test derivation tool, namely Autolink. Each trace from this set can be transformed into a correspondent MSC. A function which will implement such a transformation can easily be incorporated in the *Unfold* module. This MSC can be the test purpose used by Autolink for deriving test cases. Such a way of building test purposes can represent an alternative way to the ones provided by the Autolink tool. In this way the work presented here can give useful insights to other test related communities.

In the next example we will show how the *Unfold* function works. This toy example will also introduce the ingredients for the data structures of the program, such as traces and marked traces.

Example Consider the automaton from Figure 9.1. This automaton has two states labelled with 0 and 1. The initial state is 0. From 0, via a the state 1 can be reached. In this state, the transition b leads to the same state and the transition y to 0. Now let us fix the cycle limit to 1.

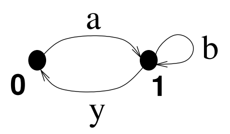


Figure 9.1: An automaton.

When applying the *Unfold* algorithm on this automaton for $l_c = 1$ the set of traces obtained is $\mathcal{F} = \{ayab, abyab\}$ (one should take $\mathcal{F}$ and its prefixes). The corresponding set of marked traces is: $\{[\langle ay\rangle]^0 a[\langle b\rangle]^1, [\langle a[\langle b\rangle]^1 y\rangle]^0 a[\langle b\rangle]^1\}$. We remind that the power of a term $[\_]^-$ is the state name. A look at the corresponding set of marked traces shows that none of these traces passes the limit l_c.

Now, once we presented possible ways of connecting the module *Unfold* of the C program to test generation tools and we illustrated how the function *Unfold* works, we can describe the data structures of the program. This will be done in the next section.

9.3 Data structures

In this section we make a short description of the data structures of the program. For a detailed description see Appendix B.1. In the *C* program a transition is represented as a structure containing the triple ⟨ *StateSource*, *Label*, *StateDestination* ⟩. A path is an array of transitions with a maximum number of locations given by the constant *MaxDepth*. A trace is represented as an array of labels. The automaton is represented as a matrix of transitions.

A marked trace is an array of elements of type *struct mt*. Each element has the following two fields: *Label* and *Cycle*. In function of the value contained by the current position of the marked trace, one of these fields will be filled: if there is a label, the *Label* field will contain its value and if there is a cycle, the *Cycle* field will be filled. The field *Cycle* is a structure itself. It is formed by the following fields: *Nr* which stores the number of times the state of the cycle is traversed; *State* which contains the state traversed by the cycle; *Elem* which stores the elements which form the iterations of the cycle. The field *Elem* is a matrix. On each row of *Elem* a marked trace corresponding to an iteration is stored. Each marked trace can contain other cycles. For representing cycles in cycles we used pointers for *Elem*.

9.4 A phone specification

Within our experiments regarding benchmarking of test generation tools, we developed together with researchers from KPN a specification for a network of phones. For working out an example for the test selection program, we derived from this specification the FSM specification of a telephone (see Figure 9.2). The specification presented is simpler than the original one, in the sense that the phone does not have the Call Waiting functionality. Despite the simplification, it is general enough to be used for highlighting the main features of the test selection program presented in this chapter.

The phone exchanges messages with the speaker and the network of phones. The convention is that when the message is received from the network the letters *np* (Network to Phone) precede the message. For example the message *np_ring_tone* is a signal sent by the network to the phone announcing that somebody is calling. When a message is sent from the phone to the network, the letters *pn* (Phone to Network) will precede the message. For example, the signal *pn_disc* is sent from the phone to the network announcing that the speaker disconnected by doing an on hook. When the signals are sent from or to the user no special sequence of letters precedes the message.

The phone has seven states. We named them in the figure with suggestive names. In the program for test selection each of these states is represented by a number between 0 and 6. The correspondence is the following: 0 corresponds to *Idle*, 1 to *Connect_Ph*, 2 to *Dialing*, 3 to *Nr_Control*, 4 to *Connected*, 5 to *Wait_Hook* and 6 to *Ringing*.

In the phone specification there are three pairs of signals which are parameterized with variables which can take many values. The pair *number(x)/pn_number(x)* has as parameter the called phone number x. The pairs *conv_in(mes)/pn_conv(mes)* and *np_conv(mes)/conv_out(mes)* have as a parameter the message exchanged which is recorded in the variable *mes*. Between pairs of signals which are parameterized with variables which can take many values we will define the label distance to perform the selection. This will be done in Section 9.6.

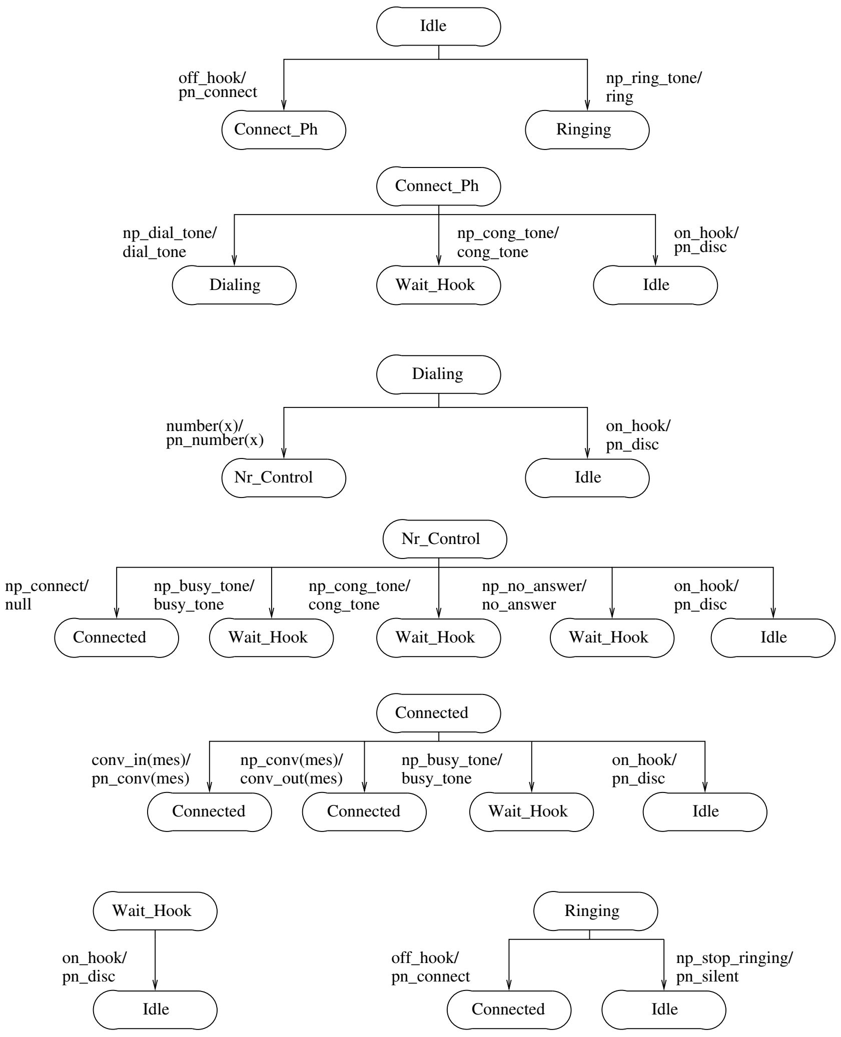


Figure 9.2: A phone specification.

9.5 An algorithm for automaton unfolding

This section describes the main algorithms of the module *Unfold*. There are two algorithms which are discussed: 1) the algorithm *Mark* which is also mentioned in Chapter 8 and which transforms a trace into a marked trace and 2) the algorithm *Unfold* which generates all the traces (the unfold tree of the automaton) which do not cycle more than a cycle limit, l_c, through the states of the automaton.

One main algorithm of *Unfold* is the function *Mark*. This algorithm transforms the path corresponding to a trace into a marked trace. This algorithm is used by the function *Unfold*. The *Mark* function is the implementation in C of the function with the same name from Chapter 8 and follows the same logic as its theoretical description. For a detailed description of it see Appendix B.2

Example The C function *Mark* can be used for computing the marked traces corresponding to the traces of the phone specification which was described in Section 9.4. For the following trace

```
off_hook/pn_connect
np_dial_tone/dialtone
```

```
number( 0043*)/pn_number( 0043*)
np_cong_tone/cong_tone
on_hook/pn_disc
off_hook/pn_connect
np_dial_tone/dial_tone
number(00351*)/pn_number(00351*)
np_connect/null
np_conv(Hello)/conv_out(Hello)
np_busy_tone/busy_tone
```

which has the following path

```
0 off_hook/pn_connect
1 np_dial_tone/dialtone
2 number( 0043*)/pn_number( 0043*)
3 np_cong_tone/cong_tone
5 on_hook/pn_disc
0 off_hook/pn_connect
1  np_dial_tone/dial_tone
2 number(00351*)/pn_number(00351*)
3 np_connect/null
4 np_conv(Hello)/conv_out(Hello)
4 np_busy_tone/busy_tone 5
```

the corresponding marked trace which is generated with *Mark* is:

```
[< off_hook/pn_connect
   np_dial_tone/dialtone
   number( 0043*)/pn_number( 0043*)
   np_cong_tone/cong_tone
   on_hook/pn_disc >]^{0}
   off_hook/pn_connect
np_dial_tone/dial_tone
number(00351*)/pn_number(00351*)
np_connect/null
[< np_conv(Hello)/conv_out(Hello) >]^{4}
np_busy_tone/busy_tone
```

Now we are going to shortly describe the algorithm *Unfold*. For a detailed description of it see Appendix B.2. As we mentioned, the objective of *Unfold* is to generate all the traces which do not cycle more than a cycle limit through the states of the automaton. Each path is built by recursive calls of *Unfold*. After a path is constructed, its corresponding trace and marked trace, both of them are saved in an output file. Then, *Unfold* starts to construct a new path. When a path is constructed, *Unfold* checks that the path do not cycle more than a cycle limit through the states of the automaton by transforming the path in its correspondent marked trace (the function used is *Mark*) and checking all the cycles symbols ([_]$^-$) of the marked trace. Constructing the path by recursive calls is easy and convenient, because the recursive calls of the function keeps tracks of the labels and the states of the automaton already visited.

There are more ways to improve *Unfold* based on current techniques used for state space exploration. For example, the Spin tool [Hol91] uses the 'bitstate hashing' [Hol97] algorithm for state space exploration. One bit is reserved for a state in a vector to store the information whether the state is visited or not. A *hash* function is used to map the states to the positions in the vector. For *Unfold*, instead of transforming the trace into a marked trace and compute whether the cycle limit is exceeded or not, a similar technique for keeping track of the states visited can be used: only the states of the cycles together with their counters will be stored into a vector. The counters will be checked whether they exceed the cycle limit or not. The repeated transformation into a marked trace will be avoided, resulting into a decrease of the complexity of the algorithm. A second improvement will be to use a *hash* function for mapping the states of the automaton to the positions in a vector. This will result in a decrease of the size of the vector. Of course, as the *hash* function is not injective, the tester must accept that two different states may have been mapped to the same position (so there is a low probability that a cycle is cut off too early). Another interesting technique is used by Caesar/Aldebaran [FGK$^+$96]. One option of this tool is to use a BCG (Binary-Coded Graphs) [Gar98] for representing a labelled transition system. The BCG format uses a binary representation together with a minimization technique resulting in a much smaller memory size (up to 20 time) for storing a labelled transition system. Adapting *Unfold* to work with a binary representation of an automaton is another way of improving the algorithm. When *Unfold* deals with large automata (this might be the case if *Unfold* is used, for example, for automata derived from LOTOS specifications which can have large sizes), using BCG formats is a convenient way to store a large automaton in a small amount of memory. These are some options for improving *Unfold*.

Example The outcome of executing the function *Unfold* on the automaton of the phone specification described in Section 9.4 is shown below (the first two traces).

```
Trace Nr. 1
off_hook/pn_connect
np_dial_tone/dialtone
number( 0043*)/pn_number( 0043*)
np_cong_tone/cong_tone
on_hook/pn_disc
off_hook/pn_connect
np_dial_tone/dialtone
number( 0043 *)/pn_number( 0043*)
np_cong_tone/cong_tone

Marked Trace
[< off_hook/pn_connect
    np_dial_tone/dialtone
    number( 0043 *)/pn_number(0043*)
    np_cong_tone/cong_tone
    on_hook/pn_disc >]^{0}
off_hook/pn_connect
np_dial_tone/dialtone
number( 0043 *)/pn_number( 0043*)
np_cong_tone/cong_tone


Trace Nr. 2
off_hook/pn_connect
np_dial_tone/dialtone
number( 0043 *)/pn_number( 0043*)
np_cong_tone/cong_tone
```

```
on_hook/pn_disc
off_hook/pn_connect
np_dial_tone/dialtone
number( 0043 *)/pn_number( 0043*)
np_busy_tone/busy_tone

Marked Trace
[< off_hook/pn_connect
   np_dial_tone/dialtone
   number( 0043 *)/pn_number( 0043*)
   np_cong_tone/cong_tone
   on_hook/pn_disc >]^{0}
off_hook/pn_connect
np_dial_tone/dialtone
number( 0043 *)/pn_number( 0043*)
np_busy_tone/busy_tone
```

The cycle limit *LC* was fixed in this case to 1. As it can be easily seen, none of the cycle exceeds this limit. When we considered only one message and one phone number, for *LC* being 1 the number of traces generated with *Unfold* was 200. Increasing *LC* to 2 modified the generation to 17168 traces; for *LC* equaling 3, the number of traces was 3.7 millions. When the cardinality of the set of phones numbers increased to 13 and the cardinality of the set of messages was 3 *Unfold* generated larger sets of traces. For *LC* being 1 the cardinality of the set of traces was 34776 and for *LC* being 2 the number was larger than 36 millions. It can be observed that selecting certain values of the parameters (by using *Reduction*) is important. This will decrease the numbers of values considered, and consequently, the number of traces generated will also decrease. About *Reduction* we will discuss in the next section. The function *Unfold* can be used for generating sets of traces which can represent test purposes used by TorX for generating tests. Using a similar style as for the function *Unfold*, the TorX algorithm itself can be modified such that, when generating and executing a trace on-the-fly, the trace generated will not exceed a cycle limit. This represents another alternative for incorporating the function *Unfold* into TorX. Moreover, as we explained in Section 9.3, the same set of traces transformed in MSCs can be used as a set of test purposes for Autolink. This can be an alternative way of obtaining test purposes, next to the ones offered by Autolink.

9.6 Algorithms for distance computation

In this section we present the functions from the module *Distance*. As shown by its name, the functions defined here deal with distances. Moreover, because the *Reduction* heuristic is done by using a label distance, we will also discuss this heuristic in this section.

The general formula of the trace distance is given in Definition 8.7.2. Using the same names as in the definition, the C functions *AtomicDistance*, *EditDistance*, *EditDistanceWeighted* are used to compute the trace distance d between two marked traces. Because these functions strictly follow the definition we need not to present them in this section. An interesting application of the trace distance d is in regression testing. We will discuss this at the end of this section.

The label distance is not defined in the general theory from Chapter 8. In the general definition, the label distance is considered to be a parameter of the distance d. In our C program we consider it also to be one of the parameters of the distance d. Therefore if one wants to use this program it is necessary to define one's own distance.

There are many ways to define a label distance by using different approaches. For example one can think of defining a label distance suitable for a boundary value analysis. Another label distance can be suitable for a random selection of a number of values. Considering the label distance as a parameter of the program leaves freedom to use the *Distance* module in a convenient way.

We will give some examples of how a label distance can be constructed. We will define a label distance for a network of telephones which have specifications as described in Section 9.4. Defining a label distance for such a telephone network will give useful insights for defining other label distances for other applications. Together with the label distance we will present also the *Reduction* heuristic applied to the labels of the telephone's transition system.

For a phone, there are two situations in which signals have parameters which can take different values: 1) the pair of signals *number(x)/pn_number(x)* has as a parameter the phone number x and 2) the pairs of signals *conv_in(mes)/pn_conv(mes)* and *np_conv(mes)/conv_out(mes)* have as a parameter the text of the message *mes*. We will show how to define a label distance first for the phone numbers x and after that for the messages *mes*. For all other non-trivial cases the label distance is 1 because all the other labels are necessary for testing different behaviours of the phone (for identical labels the label distance is 0).

A phone number is formed by: 1) a country prefix; 2) a city prefix and 3) a local phone number. We assume that the goal is to test phone connections between different countries from the European Union and different cities from the Netherlands. We restrict the domain to a certain connected subset of the countries from the European Union: Austria, Belgium, Denmark, France, Germany, Italy, Luxembourg, The Netherlands, Portugal and Spain (see Figure 9.3). The names are given in alphabetical order. The prefixes of the countries are given in the following table.

Austria	Belgium	Denmark	France	Germany	Italy	Luxembourg	Netherlands	Portugal	Spain
0043	0032	0045	0033	0049	0039	00352	0031	00351	0034

When replacing x with a phone number from one of these countries the digits of the country prefix will be present on the first positions of the phone number (which is represented as a char-string). We reserved the first 5 characters for a prefix, because the longest prefix has 5 numbers (for Luxembourg or Portugal). For the rest of the countries the first digit is a space. For example, for France the prefix is represented as ' 0033'. The country prefixes are kept in the C program by the global variable *Prefix_Country*.

As we mentioned a phone number is built by three parts: the country prefix, the city prefix and the local number. This decomposition is reflected in the label distance in the following way: we built the label distance as a sum of atomic weighted distances. Let us consider two phone numbers x and x' which have the country prefixes *PCtr* and *PCtr'*; their city prefixes are given by *PC* and *PC'* and the local numbers are *PL* and *PL'*. Then

$$DlPhNr(x,x') = w_1 \times Dist_1(PCtr, PCtr') + w_2 \times Dist_2(PC, PC') + w_3 \times Dist_3(PL, PL')$$

where w_1, w_2 and w_3 are the weights; $Dist_1, Dist_2$ and $Dist_3$ are the distances between countries prefixes, city prefixes and local numbers and *DlPhNr* represents the distance between two phone numbers. We will show the computation for each of the three distances below.

For the country prefixes we considered that a natural way for the occurrence of errors is when a call is routed through different telephone networks of the different countries. For example, taking the shortest path, between the Netherlands and France a phone call should go through the Belgian telephony system. Following this reasoning, we consider the number of border crossings a good candidate for computing the distance between two country prefixes. The numbers of border crossings between all pairs of countries are given in the following table.

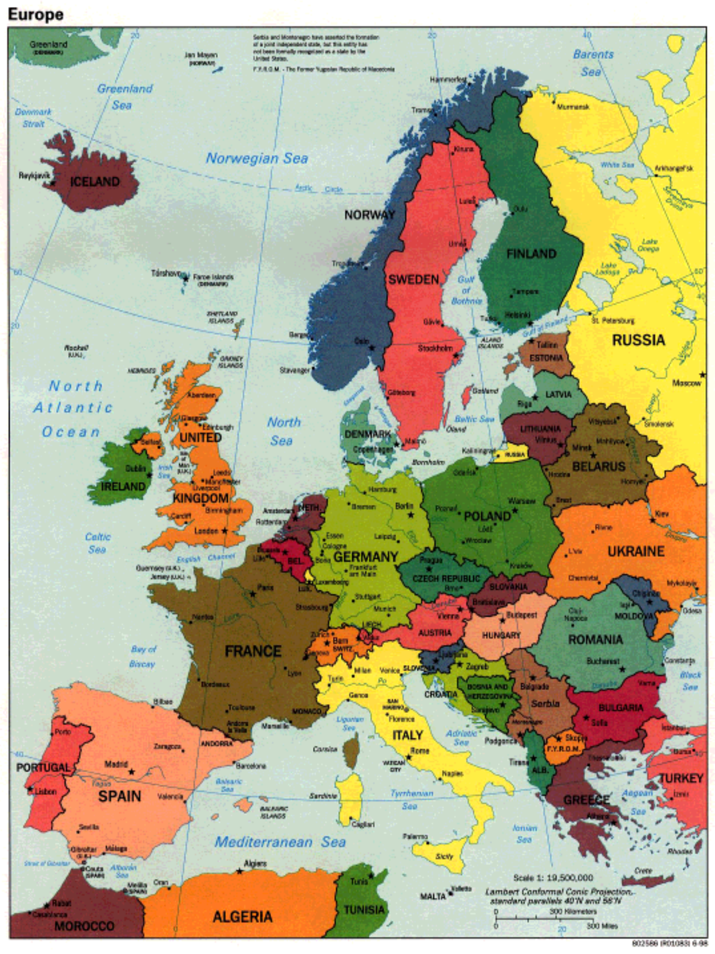


Figure 9.3: The map of Europe.

The maximal number of border crossings between two countries is 4 (between Portugal and Austria or between Portugal and Netherlands). The distance $Dist_1$ between two country prefixes is given by the number of border crossings divided by 4 (we scaled the distance back to the range [0, 1]). The values from the table above are kept in the C program by the global variable *BorderCross*.

	Austria	Belgium	Denmark	France	Germany	Italy	Luxembourg	Netherlands	Portugal	Spain
Austria	0	2	2	2	1	1	2	2	4	3
Belgium	2	0	2	1	1	2	1	1	3	2
Denmark	2	2	0	2	1	3	2	2	4	3
France	2	1	2	0	1	1	1	2	2	1
Germany	1	1	1	1	0	2	1	1	3	2
Italy	1	2	3	1	2	0	2	3	3	2
Luxembourg	2	1	2	1	1	2	0	2	3	2
Netherlands	2	1	2	2	1	3	2	0	4	3
Portugal	4	3	4	2	3	3	3	4	0	1
Spain	3	2	3	1	2	2	2	3	1	0

Next we considered cities from the Netherlands. There are three cities considered: Amsterdam, Eindhoven and Rotterdam. The prefixes of these cities are given in the following table:

Amsterdam	Rotterdam	Eindhoven
20	10	40

These prefixes are stored in the C program by the global variable *PrefixCity*.
The physical distances of the cities are given in kilometers in the following table.

	Amsterdam	Rotterdam	Eindhoven
Amsterdam	0	72	119
Rotterdam	72	0	109
Eindhoven	119	109	0

The values from this table are stored in the global variable *DistCity*. The distance between the prefixes of two considered cities from the Netherlands is computed as the physical distance divided by 119. The distance is thus scaled back to [0, 1] dividing by 119. For example, the distance between Amsterdam and Rotterdam is $\frac{72}{119}$ which is 0.60. Considering more cities from the Netherlands, the distances between their prefixes can be computed in a similar style. We assumed that within the telephony system of one country errors are more likely when the distance becomes larger. Between two cities from different countries the distance is considered to be maximal (1) because they are coming from different telephony systems, say two different clusters. For example, we want to test all the inhabitants of Germany. Testing all is impossible. Phoning from the Netherlands to two users in Germany does not make too much difference. In this case the label distance is mostly based on the distance between the country prefixes.

In the case of local numbers, we considered the numbers of the Technical University Eindhoven (TU/e for short) and Philips Research (Philips for short), both institutions located in Eindhoven. Their prefixes are given in the following table:

TU/e	Philips
247	274

These prefixes are stored in the C program by the global variable *PrefixCompany*. The physical distance between the companies are given in the following table.

	TU/e	Philips
TU/e	0	1
Philips	1	0

These values are stored in the C program in the global variable *DistCompany*. The distance between two local phone numbers is computed as the physical distance between them. Considering more local phone numbers, the distance can be computed in a similar style using the physical distance between the users. Of course, as in the case of the cities, the distance between two local numbers from two different cities is one.

For completing the picture we should say that when defining a phone number in the C program, we used the character '*' which means 'any'. For example the phone numbers ' 0033*' and ' 0045*' means a phone number from France and Denmark for which the city prefixes and the local numbers are not specified. The label distance computations between phone numbers are made in the C program by the Function *DlPhoneNumber*. For example, using ' 0033*' and ' 0045*' which represent two phone numbers from France and Denmark the returned value is:

```
DlPhoneNumber(' 0033*', ' 0045*') = 0.6
```

This computation is made as follows. The weights considered in the C program are: $w_1 = 0.8$, $w_2 = 0.18$ and $w_3 = 0.02$. The values of the weights are stored in the C program by the variables *ScaleCountry*, *ScaleCity* and *ScaleCompany*. These values reflect the assumption that the most important thing is to check the phone communication between the countries, after that between cities and after that between local numbers. For our example, the distance between the city prefixes and local numbers in this case is one. The computation is:

$$0.80 \times \frac{2}{4} + 0.18 \times 1 + 0.02 \times 1 = 0.60$$

We recall that the number 2 from the formula above represents the number of border crossings between France and Denmark. We illustrated the construction of the label distance for the phone numbers. This label distance can be used for performing *Reduction* with a given ε approximation on the set of phone number labels by computing an ε-cover of the whole set. Because the same thing can be done also for the set of messages, first we will define a label distance for messages.

Let us assume that the phones have an SMS (Short Message Service) functionality. SMS is an asynchronous mechanism for the delivery of short messages between mobile telephones. The size of a short message is small. In this context, the signals *conv_in(mes)/pn_conv(mes)* and *np_conv(mes)/conv_out(mes)* will represent the sending and the reception of a short message by the user. The synchronous specification from Figure 9.2 is not really a mobile telephone with SMS. But, signals like *conv_in(mes)/pn_conv(mes)* do appear in the label set of such a mobile telephone. Therefore, despite the differences, we can go forward with our example. We will show how to construct a label distance in a way different from the one used for the phone numbers. For messages, it can be assumed that the transmission between telephones of a short message can cause a mutilation of the message. Testing a selected set of messages will indicate whether the transmission is erroneous or not. For making this selection, a label distance is to be defined. Let us also assume that a user can choose from a limited set of words: 'Hello' translated in 6 languages. We use 'Hello' in English; 'Salut' in French; 'Hola' in Spanish; 'Ciao' in Italian; 'Hallo' in Dutch; 'Dav' in Danish.

For defining a label distance between two messages, first we ordered the messages according to how many users make use of each language: first we credited English to take first place (although some of the readers of this thesis will disagree with our point of view); the second was French and the third Spanish; the fourth was Italian; the fifth was Dutch and the sixth was Danish. This ordering is purely personal, it is not based on any statistics (any ordering can be chosen). Each message from the six gets a corresponding weight which reflects its importance.

The weights of the messages are given in the following table.

Hello	Salut	Hola	Ciao	Hallo	Dav
1	0.9	0.8	0.7	0.6	0.5

These values are stored in the C program by the variable *WeightMes*. Now the label distance between two messages is computed as the difference between their weights. For example the label distance between *Hello* and *Hallo* is computed as $\mid 1 - 0.6 \mid = 0.4$. The function which computes the label distance between two messages is called *DlMes* in the C program.

The way in which we computed the distance between messages corresponds to an *enumeration* approach for *Reduction*. In this approach the values are ordered after some rules and the first n values are selected. We ordered the values of the messages and the label distance starts to decrease for messages which are on remote positions. This label distance is suitable for an enumeration approach because selecting the first n labels will give a good covering of the whole set. In a similar style, in other applications label distances can be built.

For the phone number case, the label distance uses physical distances (expressed in kilometers or in numbers of border crossings). In this case the *Reduction* can be made by giving an ε value and computing the corresponding ε-cover of the whole set. The same thing can be done in the case of the messages. In the C program, the function *Selection* gives for a given ε approximation the subset of labels with a minimum cardinality which is an ε-cover of a whole set of labels.

Example Applying this function to the set of all phone numbers and taking $\varepsilon = 0.325$ (we will explain below the choice of this value) leads to the following 0.325-cover.

```
number( 0043 *)/pn_number( 0043*)   number( 0049*)/pn_number( 0049*)
number(00351*)/pn_number(00351*)
```

These three prefixes correspond to the following countries: ' 0043' for Austria, ' 0049' for Germany and '00351' for Portugal. The rest of the countries cluster around one of these three: Italy has one border crossing with Austria; Belgium, Denmark, France, Luxembourg and Netherlands have one border crossing with Germany and Spain has one border crossing with Portugal. The approximation of 0.325 tolerates an error produced by a border crossing. The more borders are crossed by a call, the more likely it is that an error occurs. Testing the communication between Germany and Portugal will increase confidence in the communication between Germany and France, between France and Spain and between Spain and Portugal. Choosing for example to test the communication between Denmark and Spain will lead to the testing of the communication between Denmark and Germany, between Germany and France and between France and Spain. Because the communication between Germany and France and between France and Spain was already tested when testing the communication between Germany and Portugal, the error which can occur is related to the border crossing between Germany and Denmark. A reasoning similar to this one can also be made for other pairs of countries. In this way the whole network of countries is almost tested. Because there is a tolerance for an error produced by a border cross and because the communications between the cities and the local numbers are not tested, this explains why there is still a possibility of errors which leads to a value of 0.325 of the ε approximation.

As we explained in the beginning of this section the label distance is used for the computation of the distance d between traces. A similar function as *Selection* can be made for regression testing (traditional regression testing is about running all tests over and over again). In regression testing, a set of tests is selected to be re-used later for testing IUTs. In the case of TorX the tests are execution traces. Now, using the trace distance d and an ε value, an ε-cover of tests can be computed. The ε-cover represents the most important behaviours from the whole set of execution traces which can be re-used later for testing IUTs. As it can be seen, the function *Selection* can be used not only for implementing the *Reduction* heuristic but also to give useful insights in the regression testing.

9.7 Conclusions

We made a C program which implements the theoretical work from Chapter 8 for test selection. This represents a kernel which later might be connected to TorX. We kept the program simple and we illustrated how it can be used on examples taken from the application domain of telephony.

We divided the program in two modules called *Unfold* and *Distance*. The module *Unfold* implements the *Cycling* heuristic while the module *Distance* deals with distance computations. The *Reduction* heuristic is implemented using the label distance defined in the module *Distance*.

From the module *Unfold* we presented two main functions *Mark* and *Unfold*. The function *Mark* is the implementation in C of the theoretical function with the same name from Chapter 8. Using it, the marked trace corresponding to a path is obtained. Obtaining marked traces makes the implementation of the *Cycling* heuristic and distance computations possible. The function *Unfold* can be seen as an experiment to implement the *Cycling* heuristic in a batch-oriented style. We made it plausible that this function can be effectively used in realistic domains, like telephony, for generating traces which do not exceed a cycle limit. The generated set of traces can be seen as an automatically generated test purpose used by TorX for generating test cases. Based on a similar approach, TorX can be modified to implement the cycle heuristic on-the-fly.

As we explained, the set of traces derived with *Unfold* can represent a test purpose objective for TorX. But not only for this tool such a set of traces can represent a test purpose. Let us take another test derivation tool, namely Autolink. Each trace from this set can be transformed into a corresponding MSC. A function which will implement such transformation can easily be incorporated in the *Unfold* module. This MSC can be the test purpose used by Autolink for deriving test cases. Such a way of building test purposes can represent an alternative way to the ones provided by the Autolink tool.

In the *Distance* module we worked out in detail a label distance for a phone specification. This gives useful insights of how similar label distances can be built for other applications. For the phone specification, the label distance was built for two categories of labels: the phone numbers and the messages exchanged.

The way in which we computed the label distance between messages corresponds to an enumeration approach for *Reduction*. In this approach the values are ordered after some rules and the first n values are selected. We ordered the values of the messages and the label distance starts to decrease for messages which are on remote positions. Selecting the first n labels will give a good coverage of the whole set.

For phone numbers, the label distance was built by using physical distances. In this case the *Reduction* can be made by given an ε value and compute the corresponding ε-cover of the whole set. In the C program this is made by the function *Selection* which computes for a given ε approximation the subset of labels with a minimum cardinality which is an ε-cover of a whole set of labels. Other approaches (different from the one presented in this chapter) can also be considered for constructing suitable label distances, such as the boundary value analysis.

The label distance is used for the computation of the distance d between traces. A similar function as *Selection* can be made for regression testing. In regression testing, a set of tests is selected to be re-used later for testing IUTs. In the case of TorX the tests are execution traces. Now, using the trace distance d and an ε value, an ε-cover of tests can be computed. The ε-cover represents the most important behaviours from the whole set of execution traces which can be re-used later for testing IUTs. As it can be seen, the function *Selection* can be used not only for implementing the *Reduction* heuristic but also to give useful insights in the regression testing.

We constructed a C program which forms the basis of a new module which can be later linked to TorX. Of course, really connecting this program to TorX is a first possible continuation of this work. Further research should also investigate the efficiency of the algorithms constructed and the implementation of other elements of the theory of test selection. For example, one can think of using the programming dynamic style of Wagner and Fisher ([Ste92]) for defining edit distances for creating more efficient algorithms for implementing distance computations.

Chapter 10

Conclusions

In this chapter, we draw conclusions regarding the work presented in this thesis. First, we draw general conclusions and after that we discuss opportunities for further research.

Chapter 1 integrated the research presented in the thesis in the general context of testing reactive systems. Chapter 2 presented briefly the *ioco* theory of test selection, which is a prerequisite for all chapters presented in this thesis. Also we provided a summary of the architecture and the main components of TorX, the prototype tool of the CdR project.

Chapter 3 presented a comparison of four algorithms for test derivation: TorX, TGV, Autolink and UIO algorithms. The algorithms are classified according to the detection power of their conformance relations. Since Autolink does not have an explicit conformance relation, a conformance relation is reconstructed for it. In this way the research presented here strengthens the conformance foundation of Autolink. This chapter treated only this classification criterion (it looks at the error detection power of the algorithms); other criteria such as complexity or timing are out of the scope of this research.

In this chapter we classified four known algorithms: TorX, TGV, Autolink (Telelogic/TAU) and UIO (UIOv) algorithms (Phact, Conformance Kit). The classification was made as a function of the conformance relation on which they are based, each conformance relation being expressed in terms of *ioco* theory. Also we consolidated the conformance foundation of Autolink by reconstructing an explicit conformance relation for it. This research treats only this criterion of classification (it looks at the error detection power of the algorithms); other criteria such as complexity or timing are out of the scope of this research.

From the theoretical analysis it resulted that TorX and TGV have the same detection power. Autolink has less detection power because it implements a less subtle relation than the first two (some situations exist in which the former two can detect an erroneous implementation and Autolink can not). We can also remark that for TGV and Autolink, in order to achieve their theoretical error detection capacity an infinite number of test purposes should be created, which is not always practical. For TorX one needs infinitely many test cases, but these are generated automatically which is also impractical because of resource and time limitations.

For comparing UIO and UIOv methods with Autolink, TGV and TorX one needs to assume that these tools are restricted to work on FSMs. Moreover, the restrictions implies different views regarding the presence or the absence of a null output in the output set for Autolink and *ioco* algorithms (TorX and TGV). Considering that such versions exist, it can be also concluded that in general UIO and UIOv algorithms (Phact) have less detection power than Autolink, TGV and TorX.

Chapter 4, which complements Chapter 3, presented the benchmarking experiment with the four tools TorX, TGV, Autolink and Phact on the Conference Protocol Case Study. We concentrated especially on the experiment with Autolink on the Conference Protocol because this was our main contribution in the joint effort of benchmarking. To conduct this benchmarking, the Conference Protocol has been modelled in four formal specification languages: SDL, LOTOS, PROMELA and FSM. Also, a set of concrete implementations has been constructed, some of which were injected with faults that were unknown to the person that assisted in the test process. The results have been compared with respect to the number of erroneous implementations that could be detected for each of the tools, and the time and effort that it took to test the implementations.

First we illustrated the main comparison results for Autolink and TorX. In the on-the-fly approach of TorX tests were fully automatically generated (in a random way) and executed. In the batch approach of Autolink the construction of tests needed manual assistance. Execution in the batch approach was done automatically. Both the on-the-fly approach and the batch approach were able to detect many erroneous implementations. Using the on-the-fly techniques all erroneous implementations could be detected, except for those that contained errors that simply could not be detected due to modelling choices in the specification and the choice of implementation relation. Using batch testing based on SDL fewer erroneous implementations were detected. On the one hand this is caused by the occurrence of Timeouts and on the other hand because less tests were executed due to the fact that manual assistance during test generation was needed. By deriving more test cases in the batch approach it is possible to increase the error detecting capability.

When the test purposes were generated automatically, TGV was able to detect the same amount of mutants as TorX. Phact was able to detect fewer mutants. While Autolink, TGV and TorX are able to generate larger test suites than the ones reported in this thesis, Phact reached the maximum size of its generated test suite. Therefore, Phact can not detect more mutants than the ones which it detected in the experiment. There are also strong similarities between the manual approach for TGV and Autolink. In both cases, the time needed for the manual creation of test purposes was significant. The numbers of test purposes which were created and transformed in test cases were small. For Autolink this depended also on the memory of the computers used. Consequently not all mutants could be detected (in both situations), some of them could have been found by larger test suites.

Chapter 5 presented a control technique for on-the-fly test generation through the extension of the TorX algorithm with explicit probabilities. Using these probabilities, the generated test suite can be tuned and optimized with respect to the chances of finding errors in the implementation.

We argued that in some cases the generated test suite can be optimized by adapting the values of these probabilities. Case studies gave evidence that assuming an equal distribution of chances, the TorX algorithm will sometimes yield relatively few really interesting test cases. Our calculations on the toy example of the candy machine also showed that an appropriate choice of the probabilities improves the ability to detect errors in the implementation.

An important question is, of course, whether there are heuristics which help in selecting appropriate values for the probabilities. In the case studies which we performed, the ratio between the number of inputs and the number of outputs in a test trace influenced the quality of the test cases. Therefore, we derived in this chapter the optimal values for the probabilities in the algorithm for a given ratio between the number of inputs and outputs. The proposed modification of the TorX algorithm has already been implemented.

Chapter 6 extended the theoretical work from Chapter 5, by presenting experimental results obtained with the probabilistic TorX. The experiment with the Conference Protocol case study confirms that the extension of the algorithm with explicit probabilities for its three choices (denoted as p_1, p_2 and p_3), probabilities which control the test generation, leads to improvements in the tests generated with respect to the chances of finding errors in the implementation.

The experiment gives rise to another idea on how to find the values of p_2 for which better results can be obtained. One can choose few mutants and make the curve $Length(p_2)$ for them. Then observing the values of p_2 which give points of minimum, compute the statistical mean (the average of these values) and apply the value of the mean for the rest of the mutants.

The experiment itself confirms that this idea is valid. We tested two sets of mutants. The first set indicated the values of p_2 which gave points of minimum. After that we computed the mean. Applying the mean on the second set of mutants, larger than the first one, we obtained points of minimum close to the value of the mean.

Chapter 7 extended the work presented in Chapter 5 into another direction by describing a way to compute the coverage for an *on-the-fly* test generation algorithm based on a probabilistic approach. The *on-the-fly* test generation and execution process and the development process of an implementation from a specification are viewed as stochastic processes. The probabilities of the stochastic processes are integrated in a generalized definition of coverage which can be used for expressing the detection power of a generated test suite.

The generalized formulas are instantiated for the *ioco* theory and the TorX algorithm and an example for this instantiation is worked out. When the number of test runs increases, the coverage increases towards the limit one, provided that there is a non-zero probability of **fail**. This expresses the expected property that after performing a sufficient number of test runs, the algorithm is able to detect at the end all bugs of an erroneous implementation. When decreasing the probability of detecting an error, the coverage also decreases, as expected. Computer programs can be made for the computation of the coverage, as a part of test generation tools such as TorX. Such programs needs probabilities as parameters some of which should be guessed which is not always trivial.

Chapters 5, 6 and 7 presented ways of controlling the on-the-fly test generation and execution and ways of defining coverage measures for on-the-fly algorithms. In Chapter 8, we dealt with another topic, namely test selection. By applying test selection a reduced set of tests is selected. Chapter 8 presents a coverage measure which expresses the error detection power of a reduced set of tests. We should note that there is no conflict between the two coverages formalized, the one for the on-the-fly test generation and execution, from Chapter 7, and the one for test selection, from Chapter 8. As explained in Chapter 1, the test selection is done before any test generation and execution. An intuitive example of a selection is to limit some parameter ranges of signals (from a specification which has signals with parameters, such as the Conference Protocol specification) to a small number of values. The coverage for test selection, from Chapter 8, expresses the detection power of the reduced set of tests which is chosen by selection. This detection power is compared with the set of errors which can be discovered by the whole set of tests. Now, from the set of tests selected only some tests are generated and executed by an on-the-fly algorithm (not necessarily all of them). The coverage for the test generation and execution from Chapter 7 expresses the detection power of the full test suite, assuming it could be generated and executed. In Chapter 8, the detection power is compared to the set of errors which can be discovered by the reduced set of tests. The two values obtained corresponding to the two coverages can be combined (for example by multiplying them). In this way the two coverages formalized in Chapter 7 and in Chapter 8 complement each other.

In Chapter 8 we dealt with test selection. Since exhaustive testing is in general impossible, an important step in the testing process is the development of a carefully selected test suite. Selection of test cases is not a trivial task. We proposed to base the selection process on a well-defined strategy. For this purpose, we formulated two heuristic principles: *the reduction heuristic* and *the cycling heuristic*. The first assumes that few outgoing transitions of a state show essentially different behaviour. The second assumes that the probability to detect erroneous behaviour in a loop decreases after each correct execution of the loop behaviour.

A heuristic is a general guideline for reducing test suites, which must be made more precise to be practically applicable. Especially for the cycling heuristic we had to introduce additional notation. The reason is that the cycling structure of a trace through a finite automaton must be made explicit. We introduced *marked traces* for this purpose, which enabled us to extend the work on cycle reduction by Vuong [ACV93, ACV97].

In order to introduce a notion of *coverage* for the test suites reduced by means of the above mentioned heuristics, we defined a *trace distance* on marked traces. The results of our studies can be used to effectively calculate the coverage of a test suite reduced with our techniques.

The proposed test selection technique can be compared with the existing theories in this area. In particular, these are the hypothesis theory developed by [CG97] and the trace distance theory of [ACV93, ACV97]. The hypothesis theory embodies the trace distance theory (see [CG97]), but the nice thing about trace distance theory is that it gives a measure for the degree to which a reduced set of traces approximates the original one. So we chose an approach which combines these two theories. In our approach, first the heuristics (the test hypotheses in the theory of [CG97]) are to be defined. After that, based on these heuristics a trace distance is built. This gives the possibility to make a test selection with a given ε approximation. The change of the heuristics leads to the change of the trace distance used in test selection.

Chapter 9 presented an implementation of the test selection theory described in the previous chapter. We made a C program which implements the theoretical work from Chapter 8 for test selection. This represents a kernel which later can be connected to TorX. We kept the program simple but general enough to make its use in real applications possible and illustrated this by examples taken from the application domain of telephony.

We divided the program in two modules called *Unfold* and *Distance*. The module *Unfold* implements the *Cycling* heuristic while the module *Distance* deals with distance computations. The *Reduction* heuristic is implemented using the label distance defined in the module *Distance*.

From the module *Unfold* we described the function *Unfold*. This function can be seen as an experimental function which implements the *Cycling* heuristic in a batch-oriented style. We made it plausible that this function can be effectively used in realistic domains, like telephony, for generating traces which do not exceed a cycle limit. The generated set of traces can be seen as an automatically generated test purpose used by TorX for generating test cases. But not only for this tool such a set of traces can represent a test purpose. Let us take another test derivation tool, namely Autolink. Each trace from this set can be transformed into a corresponding MSC. A function which will implement such transformation can easily be incorporated in the *Unfold* module. This MSC can be the test purpose used by Autolink for deriving test cases. Such a way of building test purposes can represent an alternative way to the ones provided by the Autolink tool.

In the *Distance* module we worked out in detail a label distance for a phone specification. This gives useful insights of how similar label distances can be built for other applications. For the phone specification, the label distance was built for two categories of labels: the phone numbers and the messages exchanged. The way in which we computed the label distance between messages corresponds to an enumeration approach for *Reduction*. In this approach the values are ordered after some rules and the first n values are selected. We ordered the values of the messages and the label distance starts to decrease for messages which are on remote positions. Selecting the first n labels will give a good covering of the whole set. For phone numbers, the label distance was built by using physical distances. In this case the *Reduction* can be made by given an ε value and compute the corresponding ε-cover of the whole set. Other approaches (different from the one presented in this chapter) can also be considered for constructing suitable label distances, such as the boundary value analyze.

We conclude this chapter by indicating directions for further research. As mentioned above, in this thesis we studied the benchmarking between several tools for test generation. The experimental results that are presented are based on a case study of a single protocol and a limited number of implementations. To obtain more valuable results the number of case studies and the number of experiments per case study should be increased. The case study in this thesis can be seen as one of the first initiatives towards such a test tool benchmarking activity.

We presented also a control technique for on-the-fly test generation through the extension of the TorX algorithm with explicit probabilities. We associated probabilities to the three choices of the algorithm. One way of continuing this research is to associate probabilities to inputs or groups of inputs and to explore this extension theoretically and practically. Some preliminary experiments in which TorX was used for testing a highway tolling system [dVBF02] shows that the quality of the tests increase when associating probabilities to groups of inputs.

We also described a test selection technique which uses heuristics and trace distances. A restricting requirement is that we assume the specification to be given as a minimal finite deterministic automaton. Some test generation tools already provide such a format, but others support general finite automata (or even non finite automata). Determinizing a finite automata may cost exponential time. In this case it would be interesting to know whether the theoretical results achieved in this thesis could be extended to non deterministic automata or even to non finite ones.

The fact that we studied only two heuristics in this thesis, does not mean that these are the only interesting heuristics. More heuristics can be defined, e.g. with respect to the general length of a trace and with respect to the uniformity of the number of outgoing transitions from a state. We embedded the two heuristics chosen in a more general framework which allows the incorporation of other heuristics.

We also presented an implementation of the test selection theory. We constructed a C program which forms the basis of a new module which can be later linked to TorX. Of course, really connecting this program to TorX is a first possible continuation of this work. Further research should also investigate the efficiency of the algorithms constructed and the implementation of other elements of the theory of test selection.

Bibliography

[AB98] Telelogic AB. *Telelogic* TAU *Documentation*. 1998.

[ACV93] J. Alilovic-Curgus and S.T. Vuong. A metric based theory of test selection and coverage. In A. Danthine, G. Leduc, and P. Wolper, editors, *Protocol Specification, Testing, and Verification*, volume XIII, pages 289–304. North-Holland, 1993.

[ACV97] J. Alilovic-Curgus and S.T. Vuong. Sensitivity analysis of the metric based test selection. In M. Kim, S. Kang, and K. Hong, editors, *Int. Workshop on Testing of Communicating Systems*, volume X, pages 200–219. Chapman & Hall, 1997.

[ADLU91] A.V. Aho, A.T. Dahbura, D. Lee, and M.Ü. Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and Rural Chinese Postman Tours. *IEEE Transctions on Communications*, 39(11):1604–1615, 1991.

[ALHH93] B. Algayres, Y. Lejeune, F. Hugonnet, and F. Hants. The AVALON project: A VALidation environment for SDL/MSC description. In O. Færgemand and A. Sarma, editors, *SDL'93 Using Objects*, pages 221–235. Elsevier Science Publishers B.V., 1993.

[All78] A. O. Allen. *Probability, Statistics, and Queueing theory*. Academic Press, Los Angeles, California, 1978.

[Arn93] A. Arnold. *Finite transition systems: semantics of communicating systems*. London, Prentice Hall, 1993.

[BB96] F. Brady and R.M. Barker. Infrastructural Tools for Information Technology and Telecomunications Conformance Testing INTOOL/GCI, Generic Compiler/Interpreter (GCI) Interface Specification, Version 2.2. *INTOOL doc. nr. GCI/NPL038v2*, 1996.

[BFdV$^+$99] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, *Int. Workshop on Testing of Communicating Systems*, volume XII, pages 179–196. Kluwer Academic Publishers, 1999.

[BFHdV01] A. Belifante, J. Feenstra, L. Heerink, and R.G. de Vries. Specification Based Formal Testing: The EasyLink Case Study. In *Progress 2001 – 2nd Workshop on Embedded Systems*, pages 73–82. STW Technology Foundation, Utrecht, The Netherlands, 2001.

[BRS$^+$00] L. Bousquet, S. Ramangalahy, S. Simon, C. Viho, A. Belinfante, and R.G. Vries. Formal test automation: the conference protocol with TGV/TorX. In H. Ural and R.L. Probert and G. von Bochmann, editors, *Proceedings of TestCom2000*, pages 221–228. Kluwer Academic Publishers, Canada, 2000.

[BRS01] P. Baker, E. Rudolph, and I. Schieferdecker. Graphical test specification – the graphical format of TTCN-3. In R. Reed and J. Reed, editors, *SDL2001: Meeting UML*, volume 2078 of *LNCS*, pages 148–167. Springer, 2001.

[BTV91] E. Brinksma, J. Tretmans, and L. Verhaard. A framework for test selection. In B. Jonsson, J. Parrow, and B. Pehrson, editors, *Protocol, Specification, Testing, and Verification*, volume XI, pages 233–248. North-Holland, 1991.

[CCI92] CCITT. *Specification and Description Language (SDL).* Recommendation Z.100. ITU-T General Secretariat, Geneva, 1992.

[CG96] O. Charles and R. Groz. Formalisation d'hypothèses pour l'evaluation de la couverture de test. In Bennani, Dsouli, Benkiran and Rafiq, editors, *Actes du Colloque Francophone sur l'Ingénierie des Protocoles (CFIP'96)*, pages 484–497, 1996.

[CG97] O. Charles and R. Groz. Basing test coverage on a formalization of test hypotheses. In M. Kim, S. Kang, and K. Hong, editors, *Int. Workshop on Testing of Communicating Systems*, volume X, pages 109–124. Chapman & Hall, 1997.

[Com91] D.E. Comer. *Internetworking with TCP/IP. Principles, protocols and architecture. Volume 1.* Prentice-Hall International, Inc., USA, 2nd edition, 1991.

[dVBF02] R. de Vries, A. Belinfante, and J. Feenstra. Automated testing in practice: The highway tolling system. In I. Schieferdecker, H. König, and A. Wolisz, editors, *Testing of Communicating Systems*, volume XIV, pages 219–234. Kluwer Academic Publishers, 2002.

[FGK$^+$96] J.C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mournier, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In T. Alur and A. Henzinger, editors, *Computer-Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 437–440. Springer–Verlag, 1996.

[FGM00] L. Feijs, N. Goga, and S. Mauw. Probabilities in the TorX test derivation algorithm. In S. Graf, C. Jard, and Y. Lahav, editors, *SAM2000 – 2nd Workshop on SDL and MSC*, pages 173–188. Col de Porte, Grenoble. VERIMAG, IRISA, SDL Forum Society, 2000.

[FGMT02] L.M.G. Feijs, N. Goga, S. Mauw, and J. Tretmans. Test selection, trace distance and heuristics. In I. Schieferdecker, H. König, and A. Wolisz, editors, *Testing of Communicating Systems*, volume XIV, pages 267–282. Kluwer Academic Publishers, 2002.

[FJJT96] G. Fernandez, J.C. Jard, C. Jéron, and T.Viho. Using on-the-fly verification techniques for the generation of test suites. In T. Alur and A. Henzinger, editors, *Computer-Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 348–359. Springer–Verlag, 1996.

[FMMvW98] L.M.G. Feijs, F.A.C Meijs, J.R. Moonen, and J.J. van Wamel. Conformance testing of a multimedia system using Phact. In A. Petrenko and N. Yevtushenko, editors, *Testing of Communicating Systems*, volume 11, pages 143–210. Kluwer Academic Publishers, 1998.

[Gar98] H. Garavel. *OPEN/CAESAR*: An open software architecture for verification, simulation, and testing. In B. Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 68–84. Lecture Notes in Computer Science 1384, Springer-Verlag, 1998.

[Gog01] N. Goga. Comparing TorX, Autolink, TGV and UIO test algorithms. In R. Reed and J. Reed, editors, *SDL2001: Meeting UML*, volume 2078 of *LNCS*, pages 379–402. Springer, 2001.

[Gog03a] N. Goga. Experimenting with the probabilistic TorX. In P.D. Curtis, editors, *Software Engineering for High Assurance System*, pages 13–21. Software Engineering Institute, Portland, Oregon, 2003.

[Gog03b] N. Goga. A probabilistic coverage for on-the-fly test generation algorithms. In M. Leuschel, S. Gruner and S. Presti, editors, *Automated Verification of Critical Systems*, pages 15–30. University of Southampton, England, 2003.

[GVZ00] H. Garavel, C. Viho, and M. Zendri. System design of a CC-NUMA multiprocessor architecture using formal specification, model-checking, co-simulation, and test generation. *Technical Report No. 0123456789*, Uniteé de recherche INRIA Rhône-Alpes, France, 2000.

[Hee98] L. Heerink. *Ins and Outs in refusal testing*. Phd. Thesis, University of Twente, Netherland, 1998.

[HFT00] L. Heerink, J. Feenstra, and J. Tretmans. Formal test automation: the conference protocol with Phact. In H. Ural, R.L. Probert and G. Bochmann, editors, *Proceedings of TestCom2000*, pages 211–220. Kluwer Academic Publishers, Canada, 2000.

[Hol91] G.J. Holzmann. *Design and Validation of Communication Protocols*. Prentice Hall, 1991.

[Hol97] G.J. Holzmann. Spate compression in Spin: Recursive indexing and compression training runs. *The 3rd Spin workshop*, Twente University, Enschede, The Netherlands, 1997.

[HT96] L. Heerink and J. Tretmans. Formal methods in conformance testing: a probabilistic refinement. In B. Baumgarten, H. Burkhardt, and A. Giessler, editors, *Ninth International Workshop in Testing and Communication System*, volume IX, pages 261–276, 1996.

[ISO89a] ISO. *Information Processing Systems, Open System Interconnection, Estelle – A Formal Description Technique Based on an Extended State Transition Model*. International Standard IS-9074. ISO, Geneva, 1989.

[ISO89b] ISO. *Information Processing Systems, Open System Interconnection, LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. International Standard IS-8807. ISO, Geneva, 1989.

[ISO92] ISO. *OSI – Open System interconnection, Information Technology – Open Systems Interconnection Conformance Testing Methodology and Framework – Part 1: General*

Concept – Part 2: Abstract Test Suites Specification – Part 3 : The Tree and Tabular Combined Notation (TTCN). International Standard ISO/IEC 9646–1/2/3. ISO, Geneva, 1992.

[IT93] ITU-T. *Recommendation Z.120: Message Sequence Charts (MSC).* Recommendation Z.120. ITU-T, Geneva, 1993.

[IT97] ITU-T. *Recommendation ITU-T Z.500 'Framework: Formal Methods in Conformance Testing*. Recommendation ITU-T Z.500. ITU-T, Geneva, 1997.

[JM99] T. Jéron and P. Morel. Test generation derived from model-checking. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 108–122. Springer, 1999.

[KGHS98] B. Koch, J. Grabowski, D. Hogrefe, and M. Schmitt. Autolink - a tool for automatic test generation from SDL specifications. *IEEE International Workshop on Industrial Strength Formal Specification Techniques (WIFT98)*, pages 114–128. IEEE Computer Society Press, Boca Raton, Florida, 1998.

[Koh78] Zvi Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill Book Company, New York, 1978.

[Kwa97] E. Kwast. *Protocol data dependencies: with an application in conformance test generation*. Phd. Thesis, University of Utrecht, Netherland, 1997.

[LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1123. IEEE Computer Society Press, 1996.

[MR97] S. Mauw and M.A. Reniers. High-level message sequence charts. In A. Cavalli and A. Sarma, editors, *Proceedings of the Eighth SDL Forum*, pages 291-306. Evry, France, 1997.

[MRS$^+$97] J.R. Moonen, J.M.T. Romijn, O. Sies, J.G. Springintveld, L.M.G. Feijs, and R.L.C. Koymans. A two–level approach to automated conformace testing of VHDL designs. In M. Kim, S. Kang, and K. Hong, editors, *Testing of Communicating Systems*, volume 10, pages 432–447. Chapman and Hall, 1997.

[Mye79] G.J. Myers. *The Art of Software Testing*. John Wiley & Sons Inc, 1979.

[SEG$^+$98] M. Schmitt, A. Ek, J. Grabowski, D. Hogrefe, and B. Koch. Autolink – puting SDL–based test generation into practice. In A. Petrenko, editor, *Proceedings of the 11th International Workshop on Testing Comunicating Systems (IWTCS'98)*, pages 227–243. Kluwer Academic Publishers, 1998.

[SKGH97] M. Schmitt, B. Koch, J. Grabowski, and D. Hogrefe. Autolink – A Tool for the Automatic and Semi-Automatic Test Generation. In A. Wolisz, I. Schieferdecker, and A. Rennoch, editors, *Formale Beschreibungstechniken für verteilte Systeme*, number 315 in GMD-Studien, St. Augustin. GI/ITG-Fachgespräch, GMD, 1997.

[Ste92] G.A. Stephen. String search. *Technical Report TR-92-gas-01*, School of Electronic Engineering Science, University College of North Wales, 1992.

[TPHT96] R. Terpstra, L. F. Pires, L. Heerink, and J. Tretmans. Testing theory in practice: a simple experiment. In Z. Brezocnik and T. Kapus, editors, *Proceedings of COST 247, Int. Workshop on Applied Formal Methods in System Design*, pages 168–183. Technical Report, University of Maribor, 1996.

[Tre96] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software – Concepts & Tools*, 17(3):103–120, 1996. Also: Technical Report No. 96–26, Centre for Telematics and Information Technology, University of Twente, The Netherlands.

[VCI90] S.T. Vuong, W.Y.L. Chan, and M.R. Ito. The UIOv–method for protocol test sequence generation. In J. Meer, L. Machert, and W. Effelsberg, editors, *International Workshop on Protocol Test Systems*, volume 2, pages 161–176. North–Holland, 1990.

[vdBKKW89] S.P. van de Burgt, J. Kroon, E. Kwast, and H.J. Wilts. The RNL Conformance Kit. In J. de Meer, L.Mackert, and W. Effelsberg, editors, *Proceeding of the 2nd International Workshop on Protocol Test Systems*, volume 2, pages 279–294. North–Holland, October 1989.

[Vra98] H.P.E. Vraken. *Design for test and debug in hardware/software systems*. Phd. Thesis, University of Eindhoven, Netherland, 1998.

[WF74] A.R. Wagner and M.J. Fisher. The string to string correction problem. In *Journal of the Association for Computer Machinery*, volume 21, pages 168–173, 1974.

[Zwi88] J. Zwiers. *Compositionality, concurrency and partial correctness: proof theories for networks of processes, and their connection*. Phd. Thesis, Eindhoven University of Technology, The Netherlands, 1988.

Appendix A

Proofs

A.1 Proof of Lemma 5.2.1

Lemma 5.2.1 *Consider an arbitrary but fixed finite trace which does not end in a final verdict. Let n be the number of inputs on the trace and p the length of the trace. Let r_l, $l = 1, 2, 3..n$ be the number of inputs which can be selected when the l-th input on the trace is selected. Let P_k, $k = n+1, 2, 3..p$ be the probability of the $(k-n)$-th output in the trace to be produced by the implementation. Then the probability P to generate this trace with the TorX algorithm is computed in the following way:*

$$P = \prod_{l=1}^{n}(\frac{1}{r_l} \times p_2) \times \prod_{k=n+1}^{p}(P_k \times p_3)$$

Proof: The signals of the trace are mapped onto the internal nodes of the behaviour tree. Therefore we will make the proof by induction on the length of the trace of the behaviour tree.

1. Basic step:
   The trace is empty. We are on the root node of the behaviour tree which has the probability 1. Because $n = 0$ and $p = 0$ in the formula of P, the probability computed with the formula of the lemma is also 1.

2. Induction step:
   By the induction hypothesis the probability for the current node is computed as

   $$P = \prod_{l=1}^{n}(\frac{1}{r_l} \times p_2) \times \prod_{k=n+1}^{p}(P_k \times p_3)$$

   Then we have to show that the probability to arrive in one of the next nodes (not the final state) is computed as

   $$P' = \prod_{l=1}^{n'}(\frac{1}{r'_l} \times p_2) \times \prod_{k=n'+1}^{p'}(P'_k \times p_3)$$

   for $p' = p+1$, suitable n' (either $n+1$ if one input is added or n if an output is added), suitable r'_l and P'_k (the values of r'_l and P'_k are given later). The situation is as in Figure A.1. As shown in the figure, v is the number of inputs.

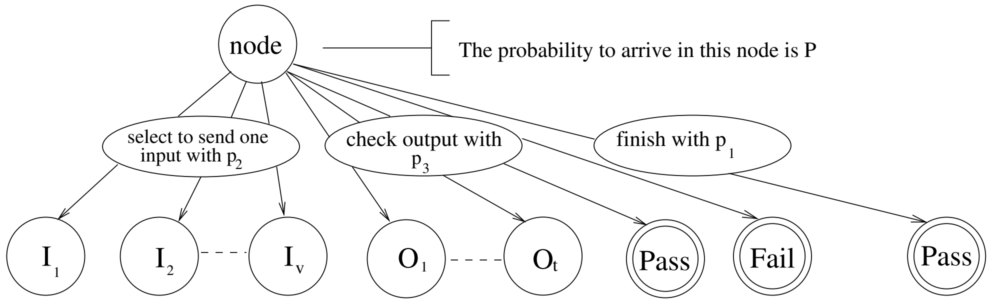

node
The probability to arrive in this node is P
select to send one input with p2
check output with p3
finish with p1
I1
I2
Iv
O1
Ot
Pass
Fail
Pass

Figure A.1: Intermediary situation.

The probability to select an input from the set of inputs in this case is $\frac{1}{v}$ (independent events). We distinguish two cases:

(a) The next node contains an input:

$$\begin{aligned} P' &= \mathrm{P}(\textit{Current node}) \times \mathrm{P}(\textit{Choice 2}) \times \mathrm{P}(\textit{Select an input from the set of inputs}) = \\ &= (\textstyle\prod_{l=1}^{n}(\frac{1}{r_l} \times p_2) \times \prod_{k=n+1}^{P}(P_k \times p_3)) \times p_2 \times \frac{1}{v} = \\ &= \textstyle\prod_{l=1}^{n'}(\frac{1}{r'_l} \times p_2) \times \prod_{k=n'+1}^{p'}(P'_k \times p_3) \end{aligned}$$

with $n' = n + 1$, $p' = p + 1$,
$r'_l = r_l$ for $l = 1, ..., n$ and $r'_{n+1} = v$
$P'_k = P_{k-1}$ for $k = n + 2, ..., p + 1$.

(b) The next node contains an output O:

$$\begin{aligned} P' &= \mathrm{P}(\textit{Current node}) \times \mathrm{P}(\textit{Choice 3}) \times \mathrm{P}(O) = \\ &= (\textstyle\prod_{l=1}^{n}(\frac{1}{r_l} \times p_2) \times \prod_{k=n+1}^{p}(P_k \times p_3)) \times p_3 \times P(O) = \\ &= \textstyle\prod_{l=1}^{n'}(\frac{1}{r'_l} \times p_2) \times \prod_{k=n'+1}^{p'}(P'_k \times p_3) \end{aligned}$$

with $n' = n$, $p' = p + 1$,
$r'_l = r_l$ for $l = 1, ..., n$,
$P'_k = P_k$ for $k = n + 1, ..., p$ and $P'_{p+1} = P(O)$. □

A.2 Proof of Proposition 7.3.1

Proposition 7.3.1 *If* $p^1_{A,s,i}(\textbf{fail}) > 0$ *then* $\lim_{m\to\infty} p^m_{A,s,i}(\textbf{fail}) = 1$

Proof: Let us put $\varepsilon = P^m_{A,s,i}(\textbf{pass})$. By assumption $\varepsilon < 1$. So:

$$\lim_{m\to\infty} P^m_{A,s,i}(\textbf{fail}) = \lim_{m\to\infty}(1 - \varepsilon^m) = 1 - \lim_{m\to\infty}(\varepsilon^m) = = 1 - 0 = 1$$

□

A.3 Proof of Proposition 7.3.2

Proposition 7.3.2 *Let $s \in$ SPECS be a specification and let $\overline{I_s}$ be the set of non-implementations of s. Let $A \in ALGS$ be an algorithm of on-the-fly test generation. Assume that an erroneous implementation occurs with a non-zero probability, formally $P_s(\overline{I_s}) > 0$. Assume that all faulty implementations that are possible can be detected, that is $\forall i \in \overline{I_s} : (p_s(i) > 0 \Rightarrow p_{A,s,i}(\textbf{fail}) > 0)$. Assume that $p_{A,s,i}(\textbf{fail}) < 1$. For positive integers m and n*

$$m < n \Rightarrow cov(A, s, m) < cov(A, s, n)$$

Proof: First we will prove the monotonicity property for the summation. Let us take $A \in$ ALGS and $s \in$ SPECS. Let $N = \overline{I_s}$ denote the set of non-implementations of s. Let us consider $i \in N$. We assume $m < n$. So:

$$(1 - p^1_{A,s,i}(\textbf{fail}))^m > (1 - p^1_{A,s,i}(\textbf{fail}))^n$$

And it follows that:

$$1 - (1 - p^1_{A,s,i}(\textbf{fail}))^m < 1 - (1 - p^1_{A,s,i}(\textbf{fail}))^n$$

Therefore:

$$p^m_{A,s,i}(\textbf{fail}) < p^n_{A,s,i}(\textbf{fail})$$

Now because $w(i) < 0$ (the implementation is erroneous) and $p_s(i) > 0$ it follows that:

$$w(i)p^m_{A,s,i}(\textbf{fail})p_s(i) > w(i)p^n_{A,s,i}(\textbf{fail})p_s(i)$$

This was for fixed but arbitrary i. Therefore it holds for all i, which means that:

$$\sum_{i\in N} w(i)p^m_{A,s,i}(\textbf{fail})p_s(i) > \sum_{i\in N} w(i)p^n_{A,s,i}(\textbf{fail})p_s(i)$$

So:

$$\lambda^m_{A,s}(N, \{\textbf{fail}\}) > \lambda^n_{A,s}(N, \{\textbf{fail}\})$$

Now because $\lambda_{A,s}(N, \{\textbf{pass}, \textbf{fail}\}) < 0$ we have that:

$$\frac{\lambda^m_{A,s}(N, \{\textbf{fail}\})}{\lambda_{A,s}(N, \{\textbf{pass}, \textbf{fail}\})} < \frac{\lambda^n_{A,s}(N, \{\textbf{fail}\})}{\lambda_{A,s}(N, \{\textbf{pass}, \textbf{fail}\})}$$

Hence $cov(A, s, m) < cov(A, s, n)$. □

A.4 Proof of Lemma 8.5.4

Lemma 8.5.4 *The width of a marked trace generated with Mark from an automaton and the widths of all its component marked traces are less than or equal to* $2m - 1$*, where m is the number of states of the automaton.*

Proof : Let us first prove this lemma for a marked trace generated with *Mark* and after that for its component marked traces. The algorithm *Mark* uses a path p which is transformed into a marked trace. The second parameter of *Mark* is the set of states Q of the automaton. If a) p does not contain a repetitive state from Q, the marked trace is $p \mid_{trace}$ (the trace corresponding to p) and its width is: $width(p \mid_{trace}) < m$ (p does not contain repetitive states). If a) does not hold, the worst case is a marked trace which goes through all states of the automaton and has a cycle in every state. This trace can be represented as $t = [\_]^{q_0} t_0 [\_]^{q_1} t_1 ... [\_]^{q_{m-2}} t_{m-2} [\_]^{q_{m-1}}$ where q_i ($i = 0, 1, ..., m - 1$) are states of the automaton and t_j ($j = 0, ..., m - 2$) are the labels of the transitions between the states. Then $width(t) = 2m - 1$. For proving the lemma for every component marked trace t', we observe that every such t' can be obtained as an application of *Mark* on a path (the path corresponding to $unfold(t')$). Therefore $width(t') \leq 2m - 1$. □

A.5 Proof of Lemma 8.5.6

Lemma 8.5.6 *The nesting depth of a marked trace generated with Mark from an automaton is less than or equal to the number of states of the automaton.*

Proof : The algorithm *Mark* starts from a path p, which is transformed into a marked trace, and the set of states Q of the automaton. Let m be the number of states of the automaton. If a) p does not contain a repetitive state from Q, the marked trace is $p \mid_{trace}$ (the trace corresponding to p) and its nesting depth is: $nesting(p \mid_{trace}) = 0 < m$. If a) does not hold, the worst case is a marked trace which has one one-state cycle at the top level and inside it one one-state cycle, and so on. This marked trace can be represented as $t = \_[\_[\_...[\_]^{q_{m-1}}...]^{q_1}]^{q_0}$ where q_i ($i = 0, 1, ..., m - 1$) are all the different states of the automaton. Then $nesting(t) = m$. □

A.6 Proof of Lemma 8.6.4

Lemma 8.6.4 *Reduction(Cycling(x)) = Cycling(Reduction(x))*

Proof: The proof is based on the following: 1) the *Reduction* function only changes the labels of a marked trace; 2) the *Cycling* function only truncates the marked traces of type $[\_]^{-}$ after l_c symbols; 3) the same marked trace is obtained if first the cycles of a trace x will be truncated and after that the labels changed ($Reduction(Cycling(x))$) or first the labels will be changed and after that the cycles truncated ($Cycling(Reduction(x))$). This means that $Reduction(Cycling(x)) = Cycling(Reduction(x))$.

Now we will elaborate the proof below. First we note that *Reduction* and *Cycling* do not change the structure of the marked traces, that is a label is mapped to a label, a pair uv to a pair $u'v'$, and a marked trace of the form $[\_]^{-}$ is mapped to something of the form $[\_]^{-}$ (this is easily shown by induction on the structure of marked traces).

The proof is by induction on the structure of marked traces. The definition of the marked traces is given by Definition 8.5.1. The grammar of marked traces is a simultaneous inductive definition of three sets: the set of marked traces, the set of non-empty marked traces and the set of non-empty sequences.

The property is trivial for the set of marked trace so it suffices to prove the same property for the set of non-empty marked traces. This is shown by proving this property for the set of non-empty marked traces and the set of non-empty sequences by simultaneous induction on the structure of non-empty marked traces and non-empty sequences.

1. if t is a non-empty marked trace because t is ut' with $u \in L$ and t' a marked trace then use the fact that the property that

   $$Reduction(Cycling(u)) = trans(u) = Cycling(Reduction((u))$$

   and it holds also for t';

2. if t is a non-empty trace because t is $[s]^q t'$ with s a non-empty sequence, $q \in Q$ and t' a marked trace then use the fact that the property holds for s and t' (for $l_c = 0$ trivial);

3. if s is a non-empty sequence because s is $\langle t \rangle$ with t a non-empty marked trace then use the fact that the property holds for t;

4. if s is a non-empty sequence because s is $\langle t \rangle s'$ with t a non-empty marked trace and s' a non-empty sequence: the property holds for t and for any term of type $\langle \_ \rangle$ of s'. If $l_c \geq \mid s \mid$ it is straight-forward that the property holds also for s. If $l_c < \mid s \mid$, based on the observation that the property holds for any term of type $\langle \_ \rangle$ of s', it holds also for s' truncated after l_c terms, and, consequently, it will hold also for s. □

A.7 Proof of Lemma 8.7.3

Lemma 8.7.3 *Let s be an automaton.*

1. *All the values of d are in the range* $[0, 1]$*, i.e.* $\forall x, y \in traces^m(s) : d(x, y) \leq 1$.

2. *The space* $(traces^m(s), d)$ *is a metric space.*

First we will prove Lemma 8.7.3 point 1).

Proof : The proof is by induction on the structure of marked traces. The grammar of marked traces is a simultaneous inductive definition of three sets: the set of marked traces, the set of non-empty marked traces and the set of non-empty sequences.

We will prove that all the values of the functions d, *AtomicDistance*, *EditDistanceWeighted* are in the range $[0, 1]$ and that all the values of the function *EditDistance* are in the range $[0, l_m]$ by simultaneous induction on the structure of non-empty marked traces and non-empty sequences.

1. if t, t' are two marked traces with $u \in L$ then for proving that d has all the values in the range $[0, 1]$ use the facts that all the values of the function *EditDistance* are in the range $[0, l_m]$ and that $d(t, t') = \frac{EditDistance(t,t')}{l_m}$ (for ϵ trivial);

2. if uv, $u'v'$ are two non-empty marked traces with $u \in L$ or $u = [\_]^-$, $u' \in L$ or $u' = [\_]^-$, v and v' are two non-empty marked traces then for proving that *EditDistance* has all the values in the range $[0, l_m]$ use the facts that 1)

$$\mathit{EditDistance}(uv_1...v_{|v|}, u'v'_1...v'_{|v'|}) \leq$$

$$\mathit{EditDistance}(u, u') + \sum_{k=1}^{\min(|v|,|v'|)} \mathit{EditDistance}(v_k, v'_k) + \sum_{k=\min(|v|,|v'|)+1}^{\max(|v|,|v'|)} 1$$

and that 2) the maximum value of a transformation from one symbol into another (the distance used is *AtomicDistance*) is 1 and that $\max(\mid v \mid, \mid v' \mid) < l_m$;

3. if u, u' are two non-empty marked traces with $u \in L$ or $u = [\_]^-$ and $u' \in L$ or $u' = [\_]^-$ then for proving that *AtomicDistance* has all the values in the range [0, 1] use the facts that d_L has this property and that the property holds for the set of non-empty sequences (the distance used is *EditDistanceWeighted*);

4. if $\langle t \rangle s$ and $\langle t' \rangle s'$ are two non-empty sequences with t and t' two non-empty marked traces and s and s' two non-empty sequences then for proving that *EditDistanceWeighted* has all the values in the range [0, 1] use the facts that the property holds for t and t' (the distance used is d) and that $\sum_{k=1}^{\infty} p_k = 1$ (similar for ($\langle t \rangle$ and $\langle t' \rangle$), ($\langle t \rangle s$ and $\langle t' \rangle$) and ($\langle t \rangle$ and $\langle t' \rangle s'$)). □

The proof of Lemma 8.7.3 point 2), i.e. $(\mathit{traces}^m(s), d)$ is a metric space, is based on the fact that the auxiliary functions *EditDistance*, *AtomicDistance* and *EditDistanceWeighted* are distances. While it is a known fact that edit distances (such as *EditDistance*) keeps the properties of metric spaces [WF74] and for *AtomicDistance* the proof is quite simple, for *EditDistanceWeighted* the proof is not that trivial. Therefore for the proof of Lemma 8.7.3 we will be helped by Proposition A.7.1 which shows that *EditDistanceWeighted* is a distance.

In Proposition A.7.1, without loss of generality, for the sake of simplicity, we consider a set of strings in place of a set of non-empty sequences. Proving that *EditDistanceWeighted* is a distance on a set of non-empty sequences is a strait-forward extension of the proof that *EditDistanceWeighted* is a distance on a set of strings.

Proposition A.7.1 *Let L be an alphabet and d_L a distance defined on this alphabet such that $\forall x, y \in L : d_L(x, y) \leq 1$. Let the cost of deleting one symbol $x \in L$, i.e. $d_L(x, \epsilon)$, be 1 and the cost of inserting one symbol $x \in L$, i.e. $d_L(\epsilon, x)$, be 1. Let p_k, with $k \in \mathbf{N}_{>0}$, be a descending series of positive numbers. Then $(L^*, \mathit{EditDistanceWeighted})$ is a metric space.*

Proof:

The function *EditDistanceWeighted* is symmetric and $\mathit{EditDistanceWeighted}(x, x) = 0$ with $x \in L^*$ because of the way in which this function was defined. What remains is to prove the triangle property. For proving the triangle property, we will use a similar style as in the literature [WF74] for the classic edit distances – non-weighted ones.

Let $a, b, c \in L^*$ be 3 strings. For proving the triangle property for these strings we will prove first the following property

$\exists$ a series of edit operations which transforms a into c, denoted as $T(a, c)$, such that:

$$\mathit{Cost}(T(a, c)) \leq \mathit{EditDistanceWeighted}(a, b) + \mathit{EditDistanceWeighted}(b, c)$$

By $Cost(T(a, c))$ we mean the sum of the edit actions which happens when transforming a into c multiplied by the corresponding weights:

$$Cost(T(a, c)) = \sum_{i=1}^{max(|a'|,|c'|)} p_i \times d_L(a'_i, c'_i)$$

where a' is a string obtained from a and c' is a string obtained from c according to the series of edit operations $T(a, c)$. More precise, if the edit operation $T_i(a, c)$ from position i has one of the following values: 1) a_j is transformed in c_k; 2) a_j is deleted or 3) c_k is inserted, with $1 \leq j \leq| a |$, $1 \leq k \leq| c |$, a_j a chracter from a and c_k a character from c, then a'_i will have one the following correspondent values: 1) a_j or 2) a_j or 3) ϵ and c'_i will have one of the following values: 1) c_k or 2) ϵ or 3) c_k.

We remind that *EditDistanceWeighted* is the minimum cost from all the costs of such series of edit operations, i.e. $EditDistanceWeighted(a, c) \leq Cost(T(a, c))$. Then if the above property holds:

$$EditDistanceWeighted(a, c) \leq EditDistanceWeighted(a, b) + EditDistanceWeighted(b, c)$$

In this way the triangle property is proved and, consequently, $(L^*, EditDistanceWeighted)$ is a metric space. What remains to be proved is that there exists a series of edit operations $T(a, c)$ such that:

$$Cost(T(a, c)) \leq EditDistanceWeighted(a, b) + EditDistanceWeighted(b, c)$$

We will construct $T(a, c)$ step by step. Let $T(a, b)$ be the series of edit operations which transforms a into b and gives the minimum cost, i.e. $EditDistanceWeighted(a, b) = Cost(T(a, b))$, and $T(b, c)$ be the series of edit operations which transforms b into c and gives the minimum cost, i.e. $EditDistanceWeighted(b, c) = Cost(T(b, c))$.

Let a' be the string obtained from a and b' the string obtained from b according to the series of edit operations $T(a, b)$. Let b'' be the string obtained from b and c' the string obtained from c according to the series of edit operations $T(b, c)$. Let $poz(b_i, b')$ be the position of the character b_i from b in the string b' and $poz(b_i, b'')$ be the position of the character b_i in the string b''. Now we have all the ingredients for starting to build $T(a, c)$.

First we will manipulate b' and b'' (and a' and c', respectively) such that at the end b' and b'' will be equal. For this we will apply the following procedure:

```
for each $b_i$ in $b$ ($1 \leq i \leq| b |$) do
  if($poz(b_i, b') \leq (poz(b_i, b'')$ then
    /* insert a number $(poz(b_i, b'') - (poz(b_i, b'))$ of $\epsilon$ in $a'$ and $b'$ between the positions $poz(b_{i-1}, b')$
    – or 1 if $i = 1$ – and $poz(b_i, b')$ */
    $d = poz(b_i, b'') - (poz(b_i, b')$;
    if(i=1) then
      $s = 0$;
    else
      $s = poz(b_{i-1}, b')$;
    /* shifting to the right */
    for $s + 1 \leq k \leq| b' |$ do
      $b'_{k+d} = b'_k$;
    for $s + 1 \leq k \leq| a' |$ do
      $a'_{k+d} = a'_k$;
    /* the insertion of $\epsilon$ */
    for $s+1 \le k \le s+d$ do
      $b'_k = a'_k = \epsilon$;
  else
    /* insert a number $(poz(b_i, b') - (poz(b_i, b'')$ of $\epsilon$ in $b''$ and $c'$ between the positions $poz(b_{i-1}, b'')$
    – or 1 if $i = 1$ – and $poz(b_i, b'')$ */
    ... /* similar as above for $b''$ and $c'$ */
```

At the end of this algorithm b' and b'' will be equal, having each character b_i from b on exact the same position.

By inserting on the same position one ϵ in a' and one ϵ in b' the structure of the transformation from a into b, i.e. $T(a,b)$, will remain unchanged, i.e. an edit operation $T_i(a,b)$ is only shifted to the right to a position $T_{i'}(a,b)$ with $i' \ge i$, the order of the edit operations is preserved and some 'null' operations 'ϵ *is transformed in* ϵ' are inserted in the series of edit operations. The same holds for b'' and c'. Taken into account that p_k is a descending series of positive numbers, we obtain that:

$$Cost(a'_f, b'_f) + Cost(b''_f, c'_f) \le Cost(a'_i, b'_i) + Cost(b''_i, c'_i) \tag{A.1}$$

where a'_i, b'_i, b''_i, c'_i are the strings a', b', b'', c' before the application of the procedure and a'_f, b'_f, b''_f, c'_f are the strings a', b', b'', c' obtained after the application of the procedure ($Cost(a', b') = \sum_{i=1}^{max(|a'|,|b'|)} p_i \times d_L(a'_i, b'_i)$). In the remainder of this proof by a', b', b'', c' we mean the strings obtained after the application of this procedure.

For getting a final transformation from a into c we need to get red of the situations in which a character b_j is transformed in ϵ in both a' and c' (see below):

poz	...	i	...
a'	...	ϵ	...
b'	...	b_j	...
b''	...	b_j	...
a'	...	ϵ	...

For solving this situation, the solution is to move the character b_j in b' and ϵ in a' to the ends of b' and a', respectively, and to shift with one position to the left the contents of b' and a': $b'_r = b'_{r+1}$, $a'_k = a'_{k+1}$, $b'_{|b'|} = b_j$ and $a'_{|a'|} = \epsilon$ with $i \le r \le \mid b' \mid -1, i \le k \le \mid a' \mid -1$. The same will be done for b'' and c'. This should be done for each such a b_j. These operations will be called *the moves of the 'nulls'*.

Based on the property that $d_L(y,z) \le 1$ for any $y, z \in L \cup \{\epsilon\}$ and that the series of p_k is descending and positive, the following holds:

$$p_k \times d_L(x,\epsilon) + p_{k+1} \times d_L(y,z) \le p_k \times d_L(y,z) + p_{k+1} \times d_L(x,\epsilon)$$

Now, if all the moves of the nulls are executed, the following holds:

$$Cost(a'_f, b'_f) + Cost(b''_f, c'_f) \le Cost(a'_i, b'_i) + Cost(b''_i, c'_i) \tag{A.2}$$

where a'_i, b'_i, b''_i, c'_i are the strings a', b', b'', c' before the moves of the nulls and a'_f, b'_f, b''_f, c'_f are the strings a', b', b'', c' after the moves of the nulls. In the remainder of this proof by a', b', b'', c' we mean the strings obtained after the moves of the nulls.

Now, the following series of edit operations $T(a,c)$ which transforms a into c can be obtained. The edit operation $T_i(a,b)$ from position i is

1. the character a_j (from a) is transformed into the character c_k (from c) iff $a'_i = a_j$ and $c'_i = c_k$;

2. the character a_j is deleted iff $a'_i = a_j$ and $c'_i = \epsilon$;
3. the character c_k is inserted iff $a'_i = \epsilon$ and $c'_i = c_k$;

Based on the property that the triangle property holds for d_L, the following holds:

$$Cost(T(a, c)) = Cost(a', c') \leq Cost(a', b') + Cost(b'', c') \tag{A.3}$$

Based on A.1, A.2, A.3 it results that:

$$Cost(T(a, c)) \leq EditDistanceWeighted(a, b) + EditDistanceWeighted(b, c)$$

Because $EditDistanceWeighted(a, b) \leq Cost(T(a, c))$ – $EditDistanceWeighted$ is the minimum of such costs – it follows that:

$$EditDistanceWeighted(a, b) \leq EditDistanceWeighted(a, b) + EditDistanceWeighted(b, c)$$

And with this it is proved that $(L^*, EditDistanceWeighted)$ is a metric space. □

Now we can give the proof of Lemma 8.7.3 point 2).

Proof : The proof is by induction on the structure of marked traces. The grammar of marked traces is a simultaneous inductive definition of three sets: the set of marked traces, the set of non-empty marked traces and the set of non-empty sequences.

We will prove that $traces^m(s), d)$ is a metric space by proving the triangle property for the functions d, $EditDistance$, $AtomicDistance$, $EditDistanceWeighted$ by simultaneous induction on the structure of non-empty marked traces and non-empty sequences. It suffices to prove only the triangle property because the properties of symmetry and that $d(x, x) = 0$ are given by definition. The case of marked traces is excluded because is trivial.

On this proof, we will be helped by the observation that all the values computed by d, $EditDistance$, $AtomicDistance$, $EditDistanceWeighted$ are in the range [0, 1] (because of the division with l_m and of the fact that $\sum_{k=1}^{\infty} p_k = 1$).

1. if uv, $u'v'$ are two non-empty marked traces with $u \in L$ or $u = [\_]^{-}$, $u' \in L$ or $u' = [\_]^{-}$, v and v' are two non-empty marked traces then for proving that $EditDistance$ has the triangle property (and, consequently, d) use the facts that the property holds for u and u' (the distance used is $AtomicDistance$) and that $EditDistance$ preserves the properties of metric spaces [WF74] (similar for (uv and u'), (u and $u'v'$));

2. if u, u' are two non-empty marked traces with $u \in L$ or $u = [\_]^{-}$ and $u' \in L$ or $u' = [\_]^{-}$ then for proving that $AtomicDistance$ has the triangle property (and, consequently, d) use the facts that (L, d_L) is a metric space, the property holds for the set of non-empty sequences (the distance used is $EditDistanceWeighted$) and that all the values of d_L and $EditDistanceWeighted$ are in the range [0, 1];

3. if $\langle t\rangle s$ and $\langle t'\rangle s'$ are two non-empty sequences with t and t' two non-empty marked traces and s and s' two non-empty sequences then for proving that $EditDistanceWeighted$ has the triangle property use the facts that the property holds for t and t' (the distance used is d) and that $EditDistanceWeighted$ preserves the properties of metric spaces – see Proposition A.7.1 (similar for ($\langle t\rangle$ and $\langle t'\rangle$), ($\langle t\rangle s$ and $\langle t'\rangle$) and ($\langle t\rangle$ and $\langle t'\rangle s'$)). □

A.8 Proof of Theorem 8.8.1

Theorem 8.8.1 *Let* $(\text{traces}^m(s), d)$ *be a metric space. Let* l_c *be the cycle limit. Let* $L_\varepsilon \subseteq L$ *be an* ε_L*-cover of* L. *Let* $\varepsilon \in [0, 1]$ *be a positive number computed in the following way:*

1. $\varepsilon = \varepsilon^z$ *with*
   (a) $\varepsilon^0 = \varepsilon_L$;
   (b) *for* $i = 1, ..., z$:
      i. $\varepsilon_c^i = \sum_{k=1}^{l_c} p_k \times (\max_{j=0,...,i-1}(\varepsilon^j)) + \sum_{k=l_c+1}^{\infty} p_k$;
      ii. $\varepsilon^i = \max_{\text{cycles}=0,...,l_m}(\frac{\text{cycles}\times\varepsilon_c^i+(l_m-\text{cycles})\times\varepsilon_L}{l_m})$.

Then the finite set T *= Ran(Reduction* $\circ$ *Cycling) of traces obtained by the application of the two heuristics on traces*$^m(s)$ *is an* ε*-cover of traces*$^m(s)$.

Proof : Let $t \in \textit{traces}^m(s)$ be a marked trace and $t' = \textit{Reduction}(\textit{Cycling}(t)) \in T$. We like to show that $d(t, t') \leq \varepsilon$. For this we observe that:

- the worst case is that all the labels from t will be changed in t' and every change of a label will give an ε_L difference;
- the worst case is that every cycle [_]$^-$ of t compared to its corresponding cycle *Reduction*(*Cycling*([_]$^-$)) from t' will give a maximum difference ε_c (we will come later to ε_c).

Then comparing the terms of t with the terms of t' sequentially from the beginning till the end we obtain that:

$$d(t, t') \leq \frac{\text{transitions} \times \varepsilon_L + \text{cycles} \times \varepsilon_c}{l_m}$$

where *transitions* is the number of transitions (or upper-level labels) in t and t' and *cycles* is the number of cycles in t and t' (because $t' = \textit{Reduction}(\textit{Cycling}(t))$, t' has the same width and a structure similar to t; if u is an upper-level label of t then $\textit{trans}(u)$ is the correspondent upper-level label of t').

Now we observe that:

- the worst case is that the marked traces t and t' have a width which equals the maximum, l_m;
- in this case the number of cycles in a marked trace can vary from 0 to l_m and the number of transitions is: transitions $= l_m-$ cycles.

Then:

$$d(t, t') \leq \max_{\text{cycles}=0,...,l_m}(\frac{\text{cycles} \times \varepsilon_c + (l_m - \text{cycles}) \times \varepsilon_L}{l_m})$$

We observe also that the worst case is that the nesting depth of t and t' will be z. Then we parameterize ε_c of the cycles with z, writing ε_c^z, and we obtain a first formula for the computation of ε^z from a series of positive numbers in the range [0, 1] (z is finite; see Section 8.5.2)

$$d(t, t') \leq \varepsilon^z = \max_{\text{cycles}=0,...,l_m}(\frac{\text{cycles} \times \varepsilon_c^z + (l_m - \text{cycles}) \times \varepsilon_L}{l_m})$$

In this formula we don't know the value of ε_c^z. We observe that:

- every cycle from t is cut in t' after l_c iterations; this means that for every cycle $u = [\langle u_1\rangle ... \langle u_k\rangle]^q$ ($k \in \mathbf{N}, q \in Q$) the marked trace t' has a corresponding cycle $u' = [\langle u'_1\rangle ... \langle u'_{\min(k,l_c)}\rangle]^q$;
- the worst case is that $k \to \infty$; in this case $\min(k, l_c) = l_c$.

Then the difference between u and u' is given by:

$$EditDistanceWeighted^1(\langle u_1\rangle ... \langle u_k\rangle, \langle u'_1\rangle ... \langle u'_{l_c}\rangle \leq \varepsilon_c^z = \sum_{k=1}^{l_c} p_k \times d(u_k, u'_k) + \sum_{k=l_c+1}^{\infty} p_k$$

But u_k and u'_k ($k = 1, ..., l_c$) are marked traces and we can compute their approximation in a similar style as we did for t and t'. The only difference is that their nesting depth can vary from 0 to $z - 1$; we will choose the maximum for ε_c^z:

$$d(u_k, u'_k) \leq \max_{j=0,...,z-1}(\varepsilon^j)$$

Now we obtain a recursive formula for the computation of the ε-cover as:
$\varepsilon^0 = \varepsilon_L$ and for $i = 1, ..., z$
$\varepsilon_c^i = \sum_{k=1}^{l_c} p_k \times (\max_{j=0,...,i-1}(\varepsilon^j)) + \sum_{k=l_c+1}^{\infty} p_k$;
$\varepsilon^i = \max_{\text{cycles}=0,...,l_m}(\frac{\text{cycles} \times \varepsilon_c^i + (l_m - \text{cycles}) \times \varepsilon_L}{l_m})$ □

A.9 Proof of Theorem 8.8.2

Theorem 8.8.2 *Let* $(\text{traces}^m(s), d)$ *be a metric space. Then for every* $\varepsilon \in [0, 1]$, *there exists a cycling limit* l_c *and a label approximation* ε_L *with* $\varepsilon_L = \sum_{k=l_c+1}^{\infty} p_k \leq \frac{\varepsilon}{2^z}$ *such that the finite set* $T =$ *Ran(Reduction* ∘ *Cycling) of traces obtained by the application of the two heuristics on* traces^m *is an* ε*-cover of traces*$^m(s)$ *and the metric space* $(traces^m(s), d)$ *is totally bounded.*

Proof : We use the results of Theorem 8.8.1 for this proof in the following way: we will like to show that $\forall i \leq z : \varepsilon^i \leq \frac{\varepsilon}{2^{z-i}}$. This has as consequence that, when $i = z : \varepsilon^z \leq \frac{\varepsilon}{2^0} \Rightarrow \varepsilon^z \leq \varepsilon$. Because T is an ε^z-cover of *traces*$^m(s)$ (cf. Theorem 8.8.1) and $\varepsilon^z \leq \varepsilon$, T is also an ε-cover of *traces*$^m(s)$ (using Definition 8.4.3).

We will prove that: $\forall i \leq z : \varepsilon^i \leq \frac{\varepsilon}{2^{z-i}}$ by induction over i.

1. *Verification*

$$\varepsilon^0 = \varepsilon_L \leq \frac{\varepsilon}{2^{z-0}} = \frac{\varepsilon}{2^z}$$

2. *Induction*

Assume that the proposition $P(i) : \forall j \leq i : \varepsilon^j \leq \frac{\varepsilon}{2^{z-j}}$ is true and prove that the proposition $P(i + 1) : \forall j \leq i + 1 : \varepsilon^j \leq \frac{\varepsilon}{2^{z-j}}$ is true. For showing that $P(i + 1)$ is true we should prove that:

$$\varepsilon^{i+1} \leq \frac{\varepsilon}{2^{z-i-1}}$$

According to Theorem 8.8.1 we have:

$$\varepsilon_c^{i+1} = \sum_{k=1}^{l_c} p_k(\max_{j=0,\dots,i}(\varepsilon^j)) + \sum_{k=l_c+1}^{\infty} p_k$$

According to the enunciation of Theorem 8.8.2 we have that $\sum_{k=l_c+1}^{\infty} p_k \leq \frac{\varepsilon}{2^z}$. Because $\sum_{k=1}^{\infty} p_k = 1$ and p_k are positive numbers, it results that $\sum_{k=1}^{l_c} p_k \leq 1$. Then:

$$\varepsilon_c^{i+1} \leq \max_{j=0,\dots,i}(\varepsilon^j) + \frac{\varepsilon}{2^z}$$

Because $P(i)$ is true we have that $\varepsilon^j \leq \frac{\varepsilon}{2^{z-j}}$ for $j \leq i$. Moreover $\frac{\varepsilon}{2^{z-j}} \leq \frac{\varepsilon}{2^{z-i}}$ because $j \leq i$. Then:

$$\varepsilon_c^{i+1} \leq \frac{\varepsilon}{2^{z-i}} + \frac{\varepsilon}{2^z}$$

Now we have $\frac{\varepsilon}{2^z} \leq \frac{\varepsilon}{2^{z-i}}$. Then:

$$\varepsilon_c^{i+1} \leq \frac{2 \times \varepsilon}{2^{z-i}} = \frac{\varepsilon}{2^{z-i-1}}$$

Now, according with Theorem 8.8.1:

$$\varepsilon^{i+1} = \max_{\text{cycles}=0,\dots,l_m} \frac{\text{cycles} \times \varepsilon_c^{i+1} + (l_m - \text{cycles}) \times \varepsilon_L}{l_m}$$

From the the enunciation of Theorem 8.8.2 we have $\sum_{k=l_c+1}^{\infty} p_k = \varepsilon_L$. We have also that $\varepsilon_c^{i+1} = \sum_{k=1}^{l_c} p_k(\max_{j=0,\dots,i}(\varepsilon^j)) + \sum_{k=l_c+1}^{\infty} p_k$. The numbers p_k and ε_j are positive. Then:

$$\varepsilon_c^{i+1} > \varepsilon_L$$

Then:

$$\varepsilon^{i+1} = \frac{l_m \times \varepsilon_c^{i+1} + (l_m - l_m) \times \varepsilon_L}{l_m} = \varepsilon_c^{i+1}$$

We showed that $\varepsilon_c^{i+1} \leq \frac{\varepsilon}{2^{z-i-1}}$. Then $\varepsilon^{i+1} \leq \frac{\varepsilon}{2^{z-i-1}}$ and in this way $P(i+1)$ is proved to be true.
□

A.10 Proof of Theorem 8.8.3

Theorem 8.8.3 *Let* $(\text{traces}^m(s), d)$ *be a metric space. Let* l_c *be the cycle limit. Then the distance* d *implements the Reduction and the Cycling heuristics.*

Proof: We use the results of Theorem 8.8.1 for this proof in the following way

1. for showing that d is implementing the *Reduction* heuristic we compute ε_r for *Ran(Reduction)* so: $\varepsilon_r = \varepsilon^z$ for $l_c = \infty$ and a given ε_L;
2. for showing that d is implementing the *Cycling* heuristic we compute ε_c for *Ran(Cycling)* so: $\varepsilon_c = \varepsilon^z$ for a given l_c and $\varepsilon_L = 0$. □

A.11 Proof of Theorem 8.8.4

Theorem 8.8.4 *Let* $(\text{traces}^m(s), d)$ *be a metric space. Let* l_c *and* l'_c *be two cycle limits* ($l_c \leq l'_c$)*. Let* $L_\varepsilon \subseteq L$ *be an* ε_L*-cover of L and* $L_{\varepsilon'} \subseteq L$ *be an* ε'_L*-cover of L such that* $L_\varepsilon \subseteq L_{\varepsilon'}$ *and* $\varepsilon_L \geq \varepsilon'_L$*. Let* T = *Ran(Reduction* ∘ *Cycling) and* T' = *Ran(Reduction* ∘ *Cycling) be the two finite sets of traces obtained by the application of the two heuristics on* $traces^m(s)$ *using* L_ε, l_c *and respectively* $L_{\varepsilon'}$, l'_c*. Then* $cov(T, traces^m(s)) \leq cov(T', traces^m(s))$*.*

Proof : According to Theorem 8.8.1, T is an ε-cover of $traces^m(s)$ and T' is an ε'-cover of $traces^m(s)$. Then $cov(T, traces^m(s)) = 1 - \varepsilon$ and $cov(T', traces^m(s)) = 1 - \varepsilon'$.

For showing that $cov(T, traces^m(s)) \leq cov(T', traces^m(s))$ we should show that $\varepsilon \geq \varepsilon'$. This can be proved by induction using the formulas of Theorem 8.8.1 for ε' and ε.

1. *Verification*

$$\varepsilon^0 = \varepsilon_L \text{ and } \varepsilon'^0 = \varepsilon'_L\text{; but } \varepsilon_L \geq \varepsilon'_L \Rightarrow \varepsilon^0 \geq \varepsilon'^0$$

2. *Induction*

Assume that the proposition $P(i) : \forall j \leq i : \varepsilon^j \geq \varepsilon'^j$ is true and prove that the proposition $P(i+1) : \forall j \leq i+1 : \varepsilon^j \geq \varepsilon'^j$ is true. For showing that $P(i+1)$ is true we should prove that

$$\varepsilon^{i+1} \geq \varepsilon'^{i+1}$$

According to Theorem 8.8.1:

$$\varepsilon_c^{i+1} = \sum_{k=1}^{l_c} p_k(\max_{j=0,\dots,i}(\varepsilon^j)) + \sum_{k=l_c+1}^{\infty} p_k$$

Because $P(i)$ is true we have that $\varepsilon^j \geq \varepsilon'^j$. Then:

$$\varepsilon_c^{i+1} \geq \sum_{k=1}^{l_c} p_k(\max_{j=0,\dots,i}(\varepsilon'^j)) + \sum_{k=l_c+1}^{\infty} p_k$$

From the enunciation of Theorem 8.8.3 we have that $l_c \leq l'_c$. Then:

$$\varepsilon_c^{i+1} \geq \sum_{k=1}^{l'_c} p_k(\max_{j=0,\dots,i}(\varepsilon'^j)) + \sum_{k=l'_c+1}^{\infty} p_k$$

But $\varepsilon'^{i+1}_c = \sum_{k=1}^{l'_c} p_k(\max_{j=0,\dots,i}(\varepsilon'^j)) + \sum_{k=l'_c+1}^{\infty} p_k$. Then:

$$\varepsilon_c^{i+1} \geq \varepsilon'^{i+1}_c$$

According to Theorem 8.8.1:

$$\varepsilon^{i+1} = \max_{\text{cycles}=0,\dots,l_m} \frac{\text{cycles} \times \varepsilon_c^{i+1} + (l_m - \text{cycles}) \times \varepsilon_L}{l_m}$$

We showed that $\varepsilon_c^{i+1} \geq \varepsilon'^{i+1}_c$. From the enunciation of Theorem 8.8.3 we have that $\varepsilon_L \geq \varepsilon'_L$. Then:

$$\varepsilon^{i+1} \geq \max_{\text{cycles}=0,\ldots,l_m} \frac{\text{cycles} \times \varepsilon'^{i+1}_c + (l_m - \text{cycles}) \times \varepsilon'_L}{l_m}$$

But $\varepsilon'^{i+1} = \max_{\text{cycles}=0,\ldots,l_m} \frac{\text{cycles} \times \varepsilon'^{i+1}_c + (l_m - \text{cycles}) \times \varepsilon'_L}{l_m}$. Then $\varepsilon^{i+1} \geq \varepsilon'^{i+1}$. In this way $P(i+1)$ is proved to be true. Now for $z = i + 1$ we have that $\varepsilon \geq \varepsilon'$. □

Appendix B

Detailed descriptions for Chapter 9

B.1 A detailed description of the data structures

For a good understanding of the functions of the program, we should give a clear view of the data structures of the program. The main structures of the program are defined for representing transitions and marked traces. Based on the transition structure, paths and automata can also be stored in the variables of the program. We kept the data structures simple so that the algorithms will be easily understood. A transition is represented as:

```
typedef struct t {
    int  StateSource;
    char Label[MaxLength];
    int  StateDestination;
} Transition;
```

In this *C* program a transition is represented as a structure containing the triple ⟨ *StateSource*, *Label*, *StateDestination* ⟩. Obviously the field *StateSource* corresponds to the state source of the transition, *Label* stores the label of the action performed and *StateDestination* stands for the destination state. The states are considered coded as natural numbers starting from 0. The labels are stored as strings of characters which have a maximum length given by the constant *MaxLength*. Using the transition data structure, a path and an automaton are declared as:

```
Transition Path[MaxDepth];
Transition Automaton[NrStates][NrMaxTran];
```

A path is an array of transitions with a maximum number of locations given by the constant *MaxDepth*. When storing a path, there should be one rule respected: the state source of the current transition should be equal to the state destination of the previous transition.

Example Let us consider the path $0 \xrightarrow{a} 1 \xrightarrow{b} 1 \xrightarrow{y} 0$ of the automaton from Figure 9.1. This path is represented in Figure B.1. The list of transitions which corresponds to this path is: $\langle 0, a, 1\rangle\langle 1, b, 1\rangle$ $\langle 1, y, 0\rangle$. The first transition is stored on the first position of the array, the second on the second position and the third on the third. The value -1 of the constant *NoState* is stored by the field *StateSource* of the transition from the fourth position of the array. This marks the end of the path. In the figure, the values $*$ from the fourth position mean any value (they are not of interest for us).

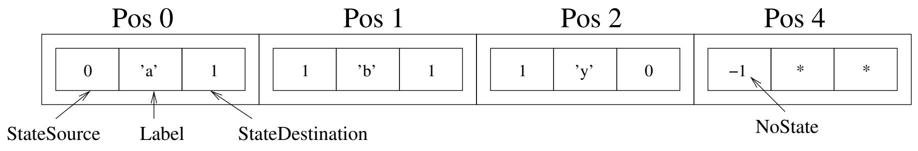


Figure B.1: A representation of $0 \xrightarrow{a} 1 \xrightarrow{b} 1 \xrightarrow{y} 0$.

From this example it can be seen that the representation of a path is a little bit redundant: the information from a state destination is copied in the state source of the next transition. For showing how to implement the test selection theory, the optimization does not play a role. For this reason we did not pay attention to it. Also, in similar situations to this one we did not work out the optimization in the program.

A trace is represented as an array of labels. Sometimes, for not duplicating the information, when we are working with a path we do not build the corresponding trace because this is formed by the labels contained by the path.

The automaton is represented as a matrix of transitions. Because we are working with automata for which the set of states is finite, there is always a bijection between the set of states of an automaton and [0..*NrStates*−1], where *NrStates* is the number of states of the automaton. Therefore we assume that the states of an automaton are labelled with numbers in the range [0..*NrStates*−1] and that the initial state is the state labelled with 0. In row i of the matrix, all the transitions with source state i of the automaton are given. The constant *NoState* indicates the end of each list of transitions.

Example Let us consider the automaton from Figure 9.1. Its representation is given in Figure B.2.

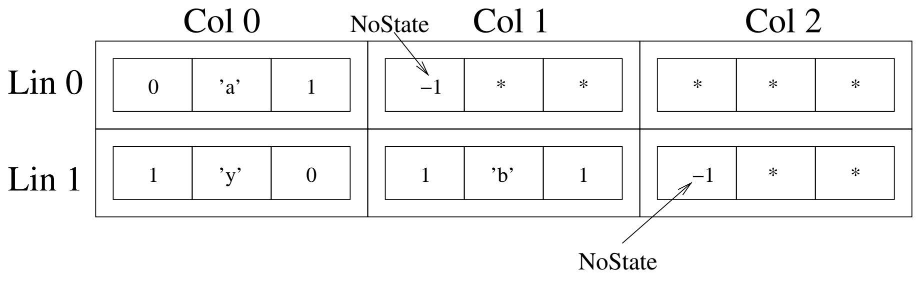


Figure B.2: The internal representation of the automaton.

There are two states of the automaton: the initial state 0 and 1. The initial state has only one transition: via a the state 1 is reached. This transition is stored in the location which has the coordinates: row 0 and column 0 in the matrix. Because there is only one transition for the state 0, the constant *NoState* is put in the *StateSource* field of the location from row 0 and column 1. This indicates the end of the list of transitions for the initial state. The transitions of the state 1 are stored in the row 1 of the matrix. The transitions are: $\langle 1, 'y', 0\rangle$ in column 0 and $\langle 1, 'b', 1\rangle$ in column 1. The constant *NoState* in column 2 marks the end of the list of transitions for the state 1.

For completing the presentation of the data structures of the program we should describe the structure corresponding to a marked trace. The data structure of a marked trace is:

```
typedef struct mt {
     char Label[MaxLength];
     struct {
            int Nr;
            int State;
            struct mt ** Elem;
      } Cycle;
} MarkedTrace[MaxDepth];
```

A marked trace is an array of elements of type *struct mt*. Each element has the following two fields: *Label* and *Cycle*. In function of the value contained by the current position of the marked trace, one of these fields will be filled: if there is a label, the *Label* field will contain its value and if there is a cycle, the *Cycle* field will be filled. The convention is the following: if the *Label* contains a value different from the constant *NoLabel* (which is the null string of characters, '') this means that a label is present on the current position of the marked trace. If the *Label* field contains the value *NoLabel*, then that means that a cycle is stored in the field *Cycle*. Thus the content of the field *Label* indicates the type of the current element of a marked trace.

The field *Cycle* is a structure itself. It is formed by the following fields: *Nr* which stores the number of times the state of the cycle is traversed; *State* which contains the state traversed by the cycle; *Elem* which stores the elements which form the iterations of the cycle.

The field *Elem* is a matrix. On each row of *Elem* a marked trace corresponding to an iteration is stored. Each marked trace can contain other cycles. For representing cycles in cycles we need to use pointers for *Elem*. Before using *Elem*, the space necessary for storing the marked traces should be allocated. After the use, this space should be made free. In this way the heap of the C program will not be filled (the heap is the space of the memory which is reserved by a C program for the dynamic variables, i.e. variables with pointers).

Example Let us consider the marked trace $a[\langle b\rangle\langle b\rangle]^1 y$ of the automaton from Figure 9.1. Its representation is given in Figure B.3. The elements of this marked trace are: 1) the label a which is stored in the field *Label* at position 0; 2) the cycle $[\langle b\rangle\langle b\rangle]^1$ which is stored in the field *Cycle* at position 1; 3) the label y which is stored in the field *Label* at position 2. The values *NoLabel* and *NoCycle* from the fourth position mark the end of the marked trace. On the position 1, the field *Label* contains the value *NoLabel*. This value indicates that a cycle is present on that position.

The cycle $[\langle b\rangle\langle b\rangle]^1$ is stored in the fields of *Cycle* as follows: *Nr* contains the value 2 which is the number of the $\langle \_ \rangle$ terms; *State* contains the value 2; *Elem* contains the values $\langle b\rangle\langle b\rangle$. For *Elem* a space of $2 \times$ *MaxDepth* locations of size *sizeof(struct mt)* should be allocated in the heap (the C function *sizeof* gives the size in octets of a data structure). The number 2 corresponds to the number of traversals of the cycle and the constant *MaxDepth* to the maximal size of a marked trace. In the first row of *Elem* the label b corresponding to the marked trace of the first iteration is stored in column 0. The values *NoLabel* and *NoCycle* from column 1 of the same row indicate the end of this marked trace. In a similar way, the second marked trace, b, is stored in the second row of *Elem*.

B.2 A detailed description of *Mark* and *Unfold*

This section describes the main algorithms of the module *Unfold*. There are two algorithms which are discussed in detail: 1) the algorithm *Mark* which is also mentioned in Chapter 8 and which transforms a trace into a marked trace and 2) the algorithm *Unfold* which generates all the traces (the unfold tree of the automaton) which do not cycle more than a cycle limit, l_c, through the states of the automaton.

There are also functions which are working as operators on paths, traces and marked traces contained by this module. For example, the function

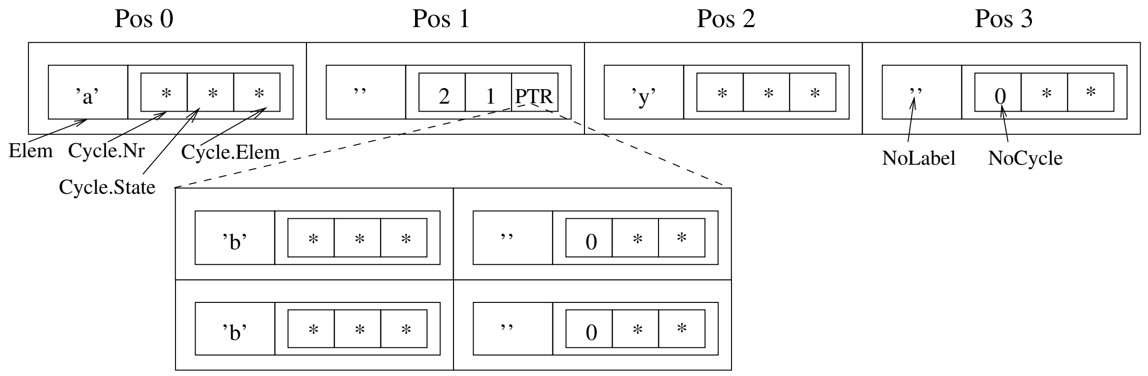


Figure B.3: A representation of a marked trace.

```
void CopyMT(MarkedTrace MTSource, MarkedTrace MTDest)
```

copies a marked trace source *MTSource* into a marked trace destination *MTDest*. Most of these operators are used by the two main algorithms mentioned above. We will explain these operators while presenting the two main algorithms. Some of these operators are also used in the module *Distance*. When describing the module *Distance*, we will also explain the operators.

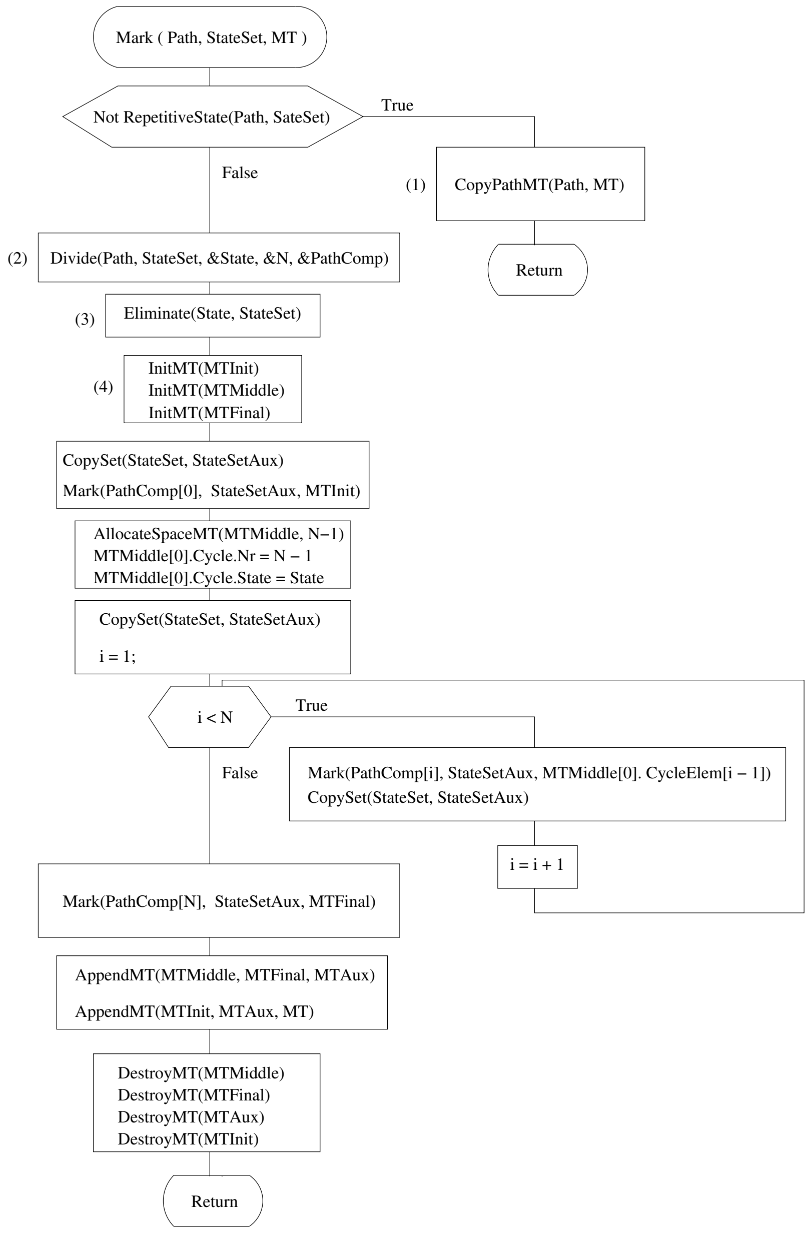
One main algorithm of *Unfold* is the function *Mark*. This algorithm transforms the path corresponding to a trace into a marked trace. This algorithm is used by the function *Unfold*. The *Mark* function is the implementation in C of the function with the same name from Chapter 8. The function was described in Chapter 8 as:

```
function Mark ($p$ : Path, $Q$ : SetStates) : MarkedTrace;
var $q$ : State;
    $p_1, ..., p_n$ : $(\epsilon+Label)(State\ Label)^*(\epsilon+State)$;
begin
  if $(\neg RepetitiveState(p, Q))$ then
(1)     return $p\ |_{trace}$;
  else
(2)     $Divide(p, Q, q, n, p_1, ..., p_n)$;
(3)     $Q = Q \setminus \{q\}$;
(4)     return $Mark(p_1 q, Q)[\langle Mark(qp_2 q, Q)\rangle ... \langle Mark(qp_{n-1} q, Q)\rangle]^q\ Mark(qp_n, Q)$;
end
```

We refer to Chapter 8 for its description. When describing the C function *Mark* we will establish the connection with the theoretical one from Chapter 8. The C function *Mark* has the following parameters.

```
void Mark(Transition Path[MaxDepth], int StateSet[NrStates], MarkedTrace MT)
```

The path p is replaced by the parameter *Path* (all the parameters of a C function which are arrays are called by reference by the function) in the C program and the set of states Q is called *StateSet*. The marked trace obtained is stored in the parameter *MT* of the C function *Mark*. The logical scheme of the C function *Mark* is depicted in Figure B.4.



Mark (Path, StateSet, MT)
Not RepetitiveState(Path, SateSet)
True
False
(1) CopyPathMT(Path, MT)
Return
(2) Divide(Path, StateSet, &State, &N, &PathComp)
(3) Eliminate(State, StateSet)
(4) InitMT(MTInit) InitMT(MTMiddle) InitMT(MTFinal)
CopySet(StateSet, StateSetAux)
Mark(PathComp[0], StateSetAux, MTInit)
AllocateSpaceMT(MTMiddle, N−1)
MTMiddle[0].Cycle.Nr = N − 1
MTMiddle[0].Cycle.State = State
CopySet(StateSet, StateSetAux)
i = 1;
i < N
True
False
Mark(PathComp[i], StateSetAux, MTMiddle[0]. CycleElem[i − 1])
CopySet(StateSet, StateSetAux)
i = i + 1
Mark(PathComp[N], StateSetAux, MTFinal)
AppendMT(MTMiddle, MTFinal, MTAux)
AppendMT(MTInit, MTAux, MT)
DestroyMT(MTMiddle)
DestroyMT(MTFinal)
DestroyMT(MTAux)
DestroyMT(MTInit)
Return

Figure B.4: The *Mark* function.

It is easy to see that the C function follows the same logic as its theoretical description. The function *RepetitiveState* has the same name in both descriptions. The operator $|_{trace}$from (1) which eliminates the states of a path is implemented by the function *CopyPathMT*. This function copies only the labels of the path *Path* in the marked trace *MT*. In this way the states are skipped by the copying process. The function *Divide* from (2) has the same name in both cases. The difference occurs at the levels of parameters. The state Q is called *State*. The contents of $p_1, ..., p_n$ are stored in the array *PathComp*. The parameter *N* from the C function equals $n - 1$ because any array in a C program is counted from 0 (*PathComp*$[i - 1]$ corresponds to p_i, $i = 1, ..., n$). The elimination of the state q from Q in the theoretical *Mark* is realized by the *Eliminate* function in the *C* program. As we explained earlier the parameter *State* stands for q and the parameter *StateSet* stands for Q. The operation (4) from the theoretical description of *Mark* is expanded in the C function. This is explained by the fact that, by using a structure with pointers for a marked trace in C, allocations of memory and freeing of memories are needed. In addition to these operations such as appending or copying are also needed in the C program.

In the case of the C function, the operation (4) starts with the initialization of three marked traces *MTInit*, *MTMiddle* and *MTFinal* by using the function *InitMT*. The marked trace *MTInit* stores the marked trace corresponding to p_1q (or *PathComp*[0] in the C program); *MTMiddle* contains the cycle corresponding to the paths qp_2q,..., $qp_{n-1}q$ (or *PathComp*[1],..., *PathComp*[$N-1$] in the C program) and *MTFinal* stands for the marked trace of qp_n (or *PathComp*[N] in the C program). The initialization consists of putting the values *NoLabel* and *NoCycle* in the fields *Label* and *Cycle.Nr* of each location of a marked trace.

After initialization, the set of states *StateSet* is copied in the variable *StateSetAux*. This is done because the content of *StateSet* is used for constructing each of the three marked traces *MTInit*, *MTMiddle* and *MTFinal*. Without using another variable, its content will be modified by a recursive call of *Mark* and would not be possible to be used later for *MTMiddle* or *MTFinal*.

After copying, the function *Mark* is recursively called with the parameters *PathComp*[0], *StateSetAux* and *MTInit*. This recursive call which stands for *Mark*(p_1q, Q) (from the theoretical description) transforms *PathComp*[0] into a marked trace which is stored in the variable *MTInit*. We recall that the states considered for cycles are the ones contained by *StateSetAux*.

For storing the cycle in *MTMiddle*, first an allocation of memory is needed for the field *Elem*. This is worked out by the function *AllocateSpaceMT*. After this, the state *State* through which the cycle is performed and the number of iterations $N-1$ are copied in the fields *Cycle.State* and *Cycle.Nr*. Again the content of *StateSet* is copied in *StateSetAux* and a loop is started by initializing the variable i with 1. The loop finishes when i reaches the value $N-1$. Within the loop, a marked trace stored in *MTMiddle*[0].*Cycle.Elem*[$i-1$] is built for each path *PathComp*[i] by a recursive call of *Mark*. The recursive call is followed by a copying action from *StateSet* to *StateSetAux*. After leaving the loop, the marked trace *MTFinal* which corresponds to *PathComp*[N] is built by another recursive call of *Mark*.

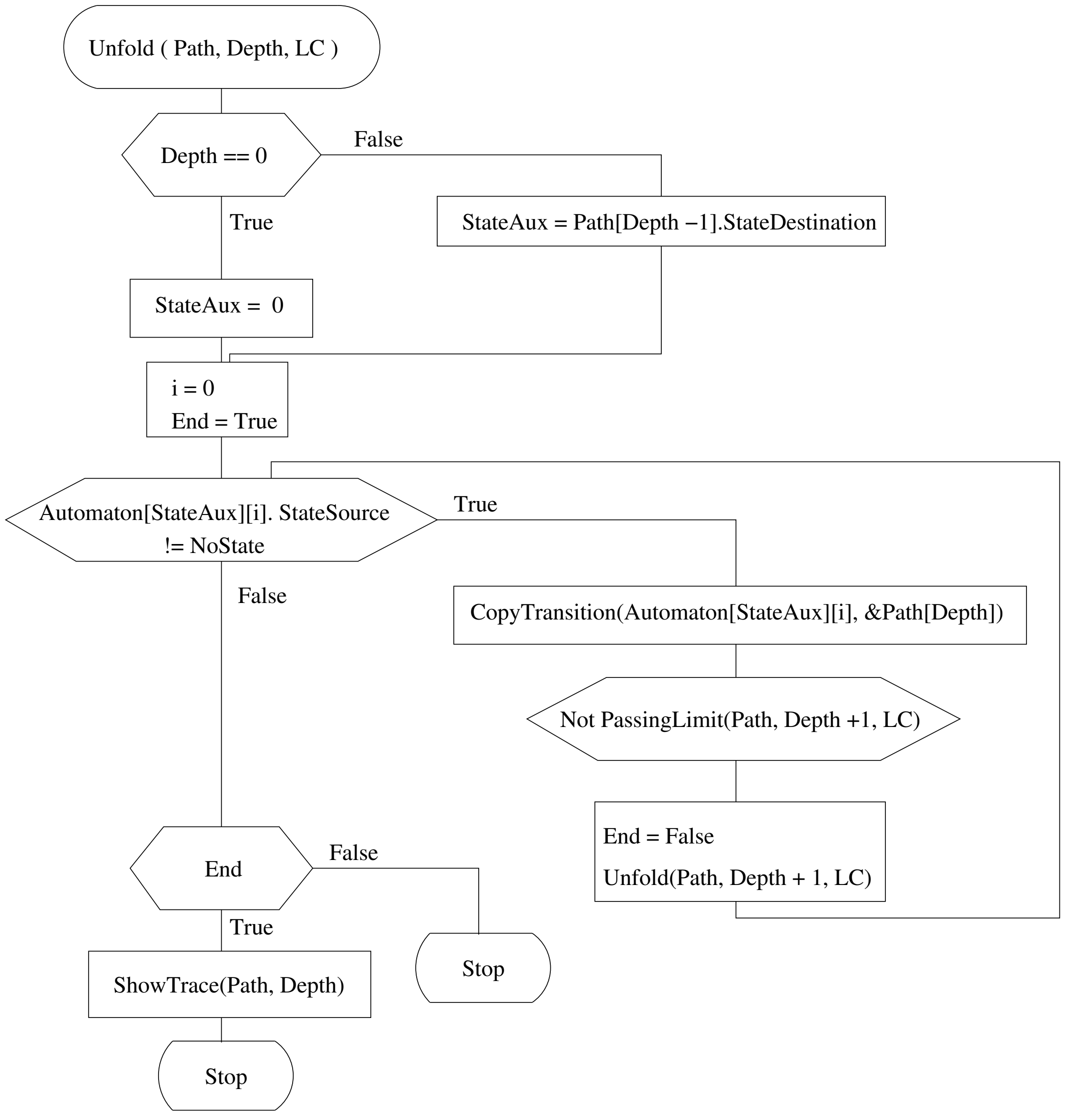
At this point all three marked traces *MTInit*, *MTMiddle* and *MTFinal* are built. They are appended by the function *AppendMT* and the marked trace *MT* is obtained. The C function *Mark* finishes by freeing the allocated memory.

Now we are going to present the algorithm *Unfold* (see Figure B.5). As we mentioned, the objective of *Unfold* is to generate all the traces which do not cycle more than a cycle limit through the states of the automaton. The algorithm *Unfold* is depicted in Figure B.5. Its parameters are:

```
void Unfold(Transition Path[MaxDepth], int Depth, int LC)
```

The transitions of the automaton are stored in the global variable *Automaton*. Because this variable does not change its value when *Unfold* is executed, it does not occur in the parameter list of *Unfold*. It occurs in the body of *Unfold*. The rest of the parameters of *Unfold* are: 1) *Path* which represents the path which is currently built by *Unfold*; 2) *Depth* which represents the level of the recursion for *Unfold* (it represents also the current length of *Path*); 3) *LC* which represents the cycling limit. Each *Path* is built by recursive calls of *Unfold*. After a *Path* is constructed, its corresponding trace and marked trace, both of them are saved in an output file. Then, the parameter *Path* can store a new path generated by *Unfold*. Constructing these paths by recursive calls is easy and convenient, because the recursive calls of the function keeps tracks of the labels and the states of the automaton already visited.

The function *Unfold* checks first the value of the parameter *Depth* which indicates whether the function is at its first call (value 0) or at a later call (a value different from 0). At the first call of



Unfold (Path, Depth, LC)
Depth == 0
False
True
StateAux = Path[Depth −1].StateDestination
StateAux = 0
i = 0
End = True
Automaton[StateAux][i]. StateSource != NoState
True
False
CopyTransition(Automaton[StateAux][i], &Path[Depth])
Not PassingLimit(Path, Depth +1, LC)
End = False
Unfold(Path, Depth + 1, LC)
End
False
True
ShowTrace(Path, Depth)
Stop
Stop

Figure B.5: The *Unfold* function.

Unfold, the local variable *StateAux* takes the value of the initial state of the automaton, 0. Otherwise the value of *StateAux* is the destination state of the last transition of *Path*. The value of *StateAux* is used in the loop which follows.

After initializing *StateAux* and before entering the loop, the variables i and *End* get the values 0 and *True*, respectively. The variable i is the counter of the loop and the variable *End* is a boolean variable which is used for indicating whether *Path* cycles more than LC times through the states of the automaton. The default value of *End* is *True*. If *Path* cycles more than LC times, the value of *End* is transformed into *False*.

In the loop, all the transitions of the state *StateAux* of the automaton are checked. The condition to leave the loop is that the location from (row *StateAux*, column i) of *Automaton* contains the value *NoState*. This value indicates the end of the list of transitions for the state *StateAux*.

A current transition of *StateAux*, contained by the location from row *StateAux*, column i of *Au-*

tomaton, is copied to *Path* at the position *Depth*. The copying is done by the function *CopyTransition*. After this operation, the function *PassingLimit* checks whether the path *Path* cycles more than *LC* times through the states of the automaton. This is done in the following way: first *Path* is transformed by *Mark* into a marked trace; after this, the counters *Nr* of all the cycles of the marked trace are checked for being larger than *LC*. If the cycling limit is not exceeded, the construction of the path *Path* is continued by a recursive call of *Unfold* and the variable *End* is set to *False*.

If for all transitions of *StateAux* the cycling limit is exceeded by *Path*, *End* keeps the value *True*. This means that there is no possibility of continuation and that the current *Path* represents one of the solutions generated by *Unfold*. The trace and the marked trace corresponding to *Path* are visualized at the standard output and at an output file. This is taken care of by the functions *ShowTrace* which shows the labels of a path and *ShowMT* which shows the content of a marked trace. The transformation of *Path* into the corresponding marked trace *MT* is done by using *Mark*.

At this point, the width and the nesting depth of *MT* are also computed. The maximum width and the maximum nesting depth of the set of all generated *MT* with *Unfold* are stored in the global variables *MaxWidth* and *MaxNestingDepth*. The values of these global variables are used for distance computations. The computations of the width and the nesting depth are not shown in Figure B.5. They are made by using the function *ComputeWidthMT* and *ComputeNestingDepthMT* which strictly follow Definition 8.5.3 and Definition 8.5.5 of the width and nesting depth of a marked trace.

Samenvatting

Het onderzoek dat in dit proefschrift gepresenteerd is, is uitgevoerd in het kader van het Côtes-de-Resyste project (afkorting: CdR). Het CdR project, gesponsord door STW, is een project op het terrein van de automatische testgeneratie. Aan dit project hebben de vogende universiteiten en bedrijven deelgenomen: Universiteit Twente, Technische Universiteit Eindhoven, Philips, KPN en Lucent. Dit proefschrift behandelt enkele onderwerpen die in het CdR project onderzocht zijn.

Een open vraag die bekeken is, is de vergelijking van bestaande tools voor automatische testgeneratie. Er bestaan verschillende tools voor automatische testgeneratie die van verschillende technieken gebruik maken. Voor een gebruiker, die een van die tools moet selecteren voor gebruik is het interessant te weten wat de mogelijkheden en beperkingen van de tools zijn. De vergelijking van dergelijke tools vanuit verschillende perspectieven, zoals snelheid en mate van foutendetectie, is van belang voor de testgemeenschap. Een ander belangrijk onderwerp is test selectie. Vaak zijn de interacties van specificaties geparameteriseerd met variabelen die vele waarden kunnen aannemen. Het meenemers van alle mogelijke parameterwaarden meenemen zal leiden tot een explosie van specificatieinteracties. Dit is slechts één reden waardoor een testgeneratie algoritme, tenminste in principe, een groot, eventueel oneindig, aantal test cases kan genereren. Omdat testexecutie beperkt dient te zijn tot een eindig aantal tests, is testselectie een belangrijk onderwerp.

In Hoofdstuk 1 wordt het onderzoek uit dit proefschrift geplaatst in de context van het testen van reactieve systemen.

Hoofdstuk 2 geeft een korte behandeling van de *ioco* theorie voor test selectie welke benodigd is voor alle hoofdstukken in dit proefschrift. Ook wordt een overzicht gegeven van de architectuur en de belangrijkste componenten van TorX, het prototype tool van het CdR project.

In Hoofdstuk 3 worden vier algoritmen voor testafleiding vergeleken: TorX, TGV, Autolink en UIO algoritmen. De algoritmen worden geklasssificeerd m.b.t. het detectievermogen van hun conformance relaties. Aangezien Autolink geen expliciete conformance relatie heeft wordt een conformance relatie geconstrueerd voor Autolink. Op deze wijze versterkt het gepresenteerde onderzoek de conformance fundering van Autolink.

Hoofdstuk 4 presenteert het benchmark-experiment met de vier tools TorX, TGV, Autolink en Phact, toegepast op het Conference Protocol. De presentatie richt zich vooral op het experiment met Autolink aangezien deze onze grootste bijdrage aan het gemeenschappelijke benchmark-experiment is.

Hoofdstuk 5 presenteert een besturingstechniek voor on-the-fly testgeneratie door het TorX algoritme uit te breiden met expliciete kansen. Gebruik makend van deze kansen kan de gegenereerde testsuite afgesteld en ge-optimaliseerd worden m.b.t. de kans om fouten in de implementatie te vinden.

Hoofdstuk 6 is een uitbreiding van het theoretische werk uit Hoofdstuk 5 met experimentele resultaten verkregen met het probabilistische TorX tool. Het experiment met het Conference Protocol bevestigt dat de uitbreiding van het algoritme met expliciete probabiliteiten die de testgeneratie besturen tot verbeteringen in de gegenereerde tests leidt m.b.t. de kans om fouten in de implementatie te vinden.

Hoofdstuk 7 gaat in een andere richting verder op het onderzoek uit hoofdstuk 5 door een manier te beschrijven om de dekkingsgraad te berekenen voor een on-the-fly testgeneratie algoritme gebaseeerd op een probabilistische aanpak. Het on-the-fly testgeneratie en executieproces en het ontwikkelingsproces van een implementatie uit een specificatie worden gezien als stochastische processen. De probabiliteiten van de stochastische processes worden geïntegreerd in een gegeneraliseerde definitie van dekkingsgraad welke gebruikt kan worden om de detectiekracht van een gegenereerde test suite uit te drukken. De gegeneraliseerde formules worden geïnstantieerd voor de ioco theorie en voor de specificatie van het testgeneratie algoritme van TorX.

Hoofdstuk 8 gaat over testselectie. Aangezien uitputtend testen in het algemeen onmogelijk is, is de ontwikkeling van een zorgvuldig samengestelde test suite een belangrijke stap in het testproces. Het selecteren van test cases is geen triviale taak. Het selectieproces zou op een goed gedefinieerde strategie gebaseerd moeten zijn. Daartoe worden twee heuristische principes geformuleerd: de reductie heuristiek en de cyclische heuristiek. De eerste veronderstelt dat maar weinig uitgaande transities van een toestand essentieel verschillend gedrag vertonen. De tweede veronderstelt dat de kans om foutief gedrag te ontdekken in een zich herhalend gedrag afneemt met iedere correcte uitvoering van dat gedrag. Deze heuristische principes worden geformuleerd en een coverage functie die dienst doet als een maat voor het foutendetectievermogen van een test suite wordt gedefinieerd. Ten behoeve hiervan worden de noties van gemarkeerde traces en een afstandsfunctie op zulke gemarkeerde traces geïntroduceerd.

Hoofdstuk 9 geeft een implementatie van de testselectie theorie uit hoofdstuk 8. Gebaseerd op een telefoniespecificatie wordt in dit hoofdstuk een voorbeeld uitgewerkt.

Hoofdstuk 10 bevat de conclusies.

Curriculum Vitae

Goga Nicolae was born on April 7, 1973 in Bucharest, Romania. In 1991 after his pre-university education (high school) at the High School Nr. 32 from Bucharest, he studied Computer Science at the Faculty of Automatics and Computer Science, Politehnica University of Bucharest, and he obtained a Master and Engineer title in Computers. From 1998 till 2002 he was employed as a PhD student at the Computing Science Department of Twente University within the project Côtes-de-Resyste (CdR), a project in the area of automatic test generation funded by the Dutch Technology Foundation STW. During these years his work was carried out at the Department of Computing Science, Technische Universiteit Eindhoven. This work has led to this thesis. Since June 2002, he is employed as a researcher at the same Department of Computing Science of the Technische Universiteit Eindhoven.

Titles in the IPA Dissertation Series

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Progam Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09